NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

A System for Validating and Executing LOTOS Data Abstractions (SVELDA)

By

Mohamed Chedly Fehri

THESIS SUBMITTED

TO THE SCHOOL OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE IN COMPUTER SCIENCE

at the

UNIVERSITY OF OTTAWA

April 1987

ISBN 0-315-53779-5

To my parents
and Kawthar

Abstract

LOTOS is an executable specification language for protocols and services currently being standardized within ISO. It is based on an extended version of Milner's Calculus of Communicating Systems (CCS) and ACT ONE Abstract Data Type (ADT) formalism. The operational semantics of ACT ONE is defined in terms of rewriting rules. After a brief introduction to LOTOS data types, we present the basic theory, procedures, and algorithms of term rewriting and conditional term rewriting, in a manner suitable for non-specialists. A principal use of rewriting systems is reasoning about the equational theories associated with a set of axioms and computing with them. In this context, the Knuth-Bendix procedure is used to obtain a decision procedure for equational theories and for constructing rewriting systems than can be used by an abstract interpreter. An extension of the Knuth-Bendix procedure to the framework of conditional axioms is presented. Two problems related to the use of this procedure; namely, its efficiency, and its early "failure" are investigated and partially solved. The development and the implementation of a System for Validating and Executing LOTOS Data Abstractions (SVELDA) are described.

SVELDA is based on the theories of term rewriting and conditional term rewriting systems. It includes an improved version of Kaplan's conditional Knuth-Bendix completion procedure. SVELDA was developed and implemented as a part of a prototype LOTOS interpreter developed at University of Ottawa during the two previous years. It can also be used by itself for experimentation. Five tasks can be performed using SVELDA: i) translate ACT ONE texts to low-level specifications that can be easily interpreted. ii) Compile the set of conditional equations of the specification to produce a complete (convergent) conditional TRS. iii) Construct a terminating conditional term rewriting system from the set of conditional equations. iv) Proving formal theorems. This involves deciding if an equation "follows" or is a consequence of a given set of conditional equations, or if an equation is "true" in a given theory. v) Evaluate expressions used as a representation of data, calculate their sorts, and check for sort correctness.

SVELDA consists of three main modules: i) A translator which translates ACT ONE text. ii) A validator which is mainly based on the conditional Knuth-Bendix procedure to test for confluence, termination, and to perform formal proofs. iii) An ADT interpreter used to evaluate expressions and to check for sort correctness.

Acknowledgments

I would like to express my foremost thanks to my supervisor, Dr. Luigi Logrippo, for his valuable time, patience and advice throughout my graduate work.

I owe a special thanks to Abdellatif Obaid, and Jean Piere Briand, friends, teammates, colleagues, thechnical collaborators, and tireless implementors, for their valuable time, and patient audiences; and thanks to the Tunisian government for its financial support under its scholarship program during my years of study.

I am very thankful to my parents who have given me an optimistic outlook, a taste for quality, and total support for whatever I want to do. Finally I would like to express my deepeast thanks to Kawthar for her patience.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Background

Since the last decade, concepts of algebraic specifications have become central in the area of software methodology and software engineering. Especially algebraic specification for abstract data types has gained considerable importance in recent years.

The formalism of algebraic specifications allows one to specify data types and software systems independent of their representation and without reference to any particular machine configuration or operating system available. With respect to semantics, they are independent of technological changes and therefore build a reliable basis for documentation and implementation. Indeed, the fact that we can write abstract specifications and know their precise meanings is a great achievement, but it is just a part of what algebraic specifications do provide: in addition, they may be viewed as an axiomatization of the theory of the data type they specify. In this respect, algebraic specifications on one hand admit formal theorem proving and allow automatization of proof of correctness and other formal properties, on the other hand they can be considered as an input to an interpreter which formally evaluates terms, thus producing normal form terms which may be serve as a representation of data. If used in these manners, the axioms of the data type specification are oriented from left to right to allow one-way replacement and thus are called Term Rewriting System (TRS).

TRSs are an interesting model of computation. They may be used to represent abstract interpreters of programming languages and to model fomula manipulating

systems used in various applications, such as program validation, automatic theorem proving, program optimization, and compilers. They were first applied in the study of word problems in algebra.

A TRS is a set of rewrite rules. Each rule is a "one-way" equation, and specifies that: if a term, or one of its subterms, overlaps with the left-hand side of the rule, the term or subterm may be "rewritten" into the right-hand side of the rule. A *normal form* for a term is a rewritten form of that term that cannot be rewritten further using any rule of the rewriting system. Two fundamental properties of TRSs are termination and confluence. A TRS is said to *terminate* (or, equivalently, to be *noetherian*) if all terms have a normal form with respect to it. A TRS is *confluent* if every term has a unique normal form (if the latter exists) independent of the order in which the rules are applied. TRSs that are both terminating and confluent are said to be *convergent* (*complete*) .

When using a set of axioms for automatic computations or for proofs, we are interested in finding a convergent TRS for that set. Unfortunately, both termination and confluence are undecidable, which gives rise to some difficulties for the problem of finding a convergent rewriting system for a given set of axioms. However, there are some known sufficient conditions for these two properties which are widely applicable and easy to automate. To prove termination, a well-known method is to exhibit a "reduction ordering" on terms such that, for each rule in the rewriting system, the left-hand side is greater than the right-hand side under that ordering.

Once termination has been established, confluence is decidable. The confluence property of a non-confluent terminating TRS can be achieved by using a well-known procedure, called the Knuth-Bendix completion procedure. This procedure consists of adding additional rules to the system in the hope of achieving confluence. All the rules added in this manner are in the equational theory of the original axioms.

So far we have only talked about classical TRSs, which are sets of rewrite rules, but sometimes it is necessary to put some condition on the usage of a rule. Hence the idea of "conditional rewrite rules". A TRS that is composed of a set of conditional rewrite rules is called Conditional Term Rewriting System (Conditional TRS). A conditional rewrite rule consists of two parts, the premises and the conclusion. The premises are conditions that must be verified prior to the use of the rule, and the conclusion is a "one-way" equation: if a term, or one of its subterms, overlaps with the left-hand side of the conclusion of a conditional rule, the term or subterm may be "rewritten" into the the right-hand side of the conclusion of the conditional rule only if the premises of that rule are satisfied. Unfortunately, normal forms are not computable for general conditional TRSs, and also the problems of termination and confluence are undecidable. However Kaplan [KAP 84b] has introduced the concept of "fair" conditional TRSs, for which the normal form is computable and the termination and confluence properties are decidable. He also introduced a Knuth-Bendix-like procedure for fair conditional TRS's.

The specification language LOTOS, a Formal Description Technique (FDT) for data communication protocols and services, is in the process of being standardized for the formal specification of Open System Interconnection (or OSI for short). It has already been the subject of several papers (see [BRI 86], [SCO 85], [CAR 84]). The definition of LOTOS is in [ISO 86]. Recall that the LOTOS language has two main components: A "control" part based on an extended version of Milner's CCS [MIL 80] (called CCS*), and a "data" component based on algebraic specifications of "Abstract Data Types" (ADTs) as in ACT ONE. ACT ONE is an experimental specification language developed by Ehrig *et al.* [EHR 85].

1.2 Motivation of the Thesis

LOTOS has been designed as an executable specification language. Executable specifications provide a "running prototype" of what is specified and thus have the advantage of being capable to be tested prior to implementation. For this purpose a LOTOS interpreter has been developed at University of Ottawa during the previous couple of years. The general structure of the interpreter is shown in Figure 1.1. It corresponds to the structure suggested in [ISO86]. The "data" part is kept separate from the "control" part. Further, both parts are implemented by first translating LOTOS code into respectively low-level ADT specification and intermediate CCS* code. Thus, the first part of the interpreter (the top four boxes) is really a compiler, which produces internal representations of the source LOTOS program. The real interpreter works on these internal representations and is made up of three parts: the CCS* interpreter, the validator, and the ADT interpreter. The validator is based on the conditional Knuth-Bendix procedure and is used to check for the completion or termination of TRSs or conditional TRSs generated from the axioms of the specification, and to prove some formal theorems. The ADT interpreter is a subordinate of the CCS* interpreter, in the sense that it evaluates ADT functions when requested. For a detailed description of the LOTOS interpreter and its usage see [BRD 86]. The shaded region in Figure 1.1 is the subject of this thesis.

The main motivations behind this thesis are : (i) study the executability and validation of the LOTOS data type definitions. This consists of using the equations and conditional equations as rewrite rules, which leads us to study the theory and algorithms related to term rewriting and conditional term rewriting systems; (ii) make some improvements to the solutions found to solve some problems related to that theory; and (iii) develop a System for Validating and Executing LOTOS

Data Abstraction (SVELDA) based on those solutions. SVELDA was designed and implemented to do the following tasks:

1. Translate ACT ONE texts existing in a LOTOS specification into a low-level data type specification, which is the union of all data types specifications introduced in the LOTOS specification.

2. Decide whether a set of conditional equations (or simple equations), can be "compiled" to produce a complete (convergent) conditional TRS (or simply a TRS). This will be done using the Knuth-Bendix-like completion procedure implementation that operates on TRSs and conditional TRSs.

3. Construct a terminating rewriting system from the set of conditional equations (or simple equations). This involves testing termination without testing confluence.

4. Reasoning about equations. This may involve deciding if an equation "follows" or is a consequence of a given set of equations, or if an equation is "true" in a given theory.

5. Evaluate expressions used as representation of data. This involves computing with equations and conditional equations considered as "conditional rewrite rules". In this manner the system is used within the LOTOS interpreter to evaluate expressions introduced in the LOTOS specification or introduced by the user at the interaction points.

6. Calculate and check sorts for given expressions.

Note that in order for task 4 and 5 to be effective one need to have a complete or a terminating conditional TRS (or simply TRS). So task 2 or task 3 must be done

first for each "non-compiled" set of conditional equations (or simple equations).

1.3 Overview of the Thesis

This thesis introduces control and data concepts in LOTOS, the basic theory and algorithms related to TRSs and conditional TRSs, and describes the design and implementation of SVELDA — the translator, the ADT interpreter, and the validator, which incorporates an improved version of the Knuth-Bendix-like completion procedure given by Kaplan.

The thesis is organized as follows: Chapter 2 introduces control and data concepts in LOTOS, and demonstrates the use of abstract data type definitions in LOTOS specifications. Chapter 3 introduces the equational and conditional equational theories, and the basic theories and algorithms related to TRSs and conditional TRSs. Chapter 4 describes the design and implementation of SVELDA: the translator, the validator together with the improvements implemented on Kaplan's Knuth-Bendix-like procedure, and the ADT Interpreter. Chapter 5 describes SVELDA's user interface. Chapter 6 summarizes the thesis, and highlights some possible areas of future work. Appendix A contains some example of use of SVELDA, while Appendix B contains the programs listing.

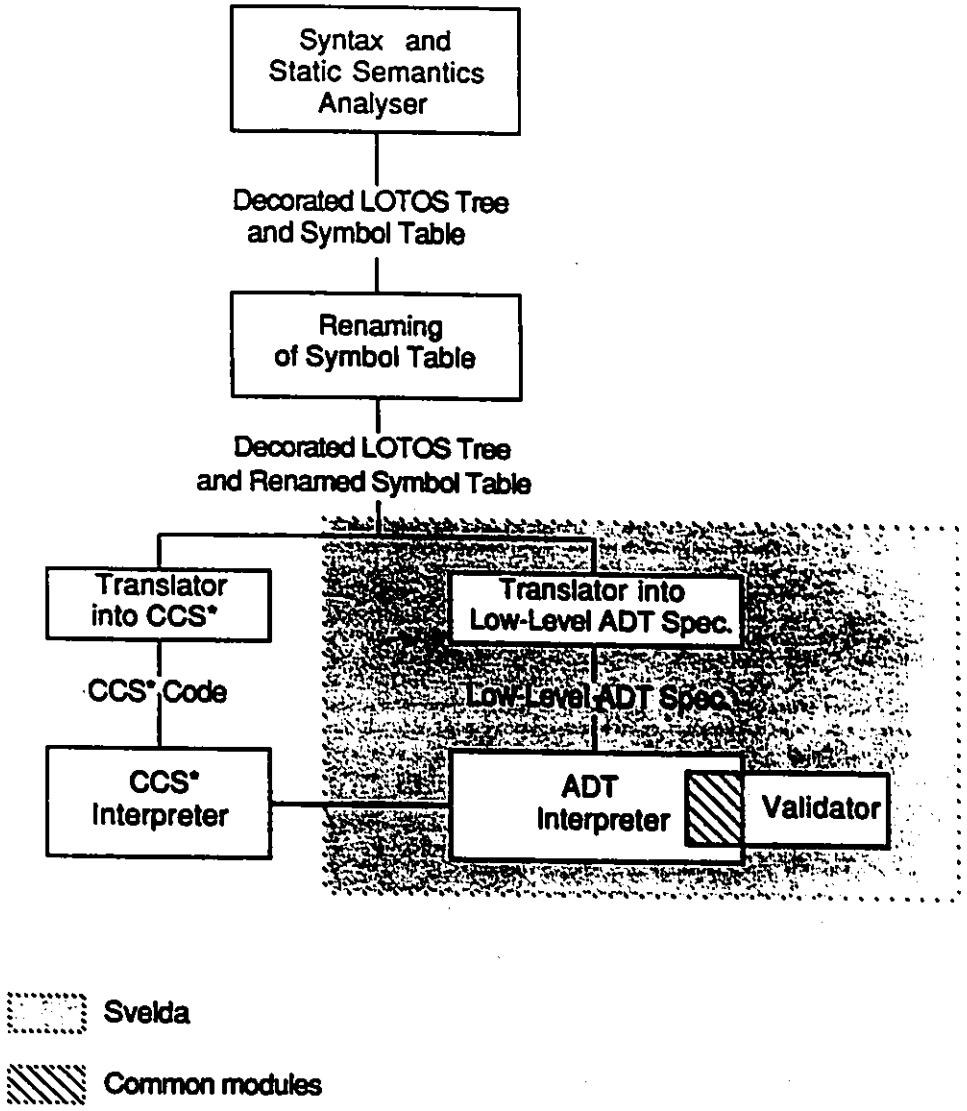

Figure 1.1 The LOTOS Interpreter Dependency Diagram

CHAPTER 2

LOTOS and its Relation to Abstract Data Types

2.1 Introduction

LOTOS is mostly based on the principles of the Calculus of Communicating Systems (CCS) of R. Milner [MIL 80], from which the representation of behavior expressions is inspired. However, it is also influenced by the related work of C.A.R Hoare, who more or less in parallel with Milner, has been developing his own theory of Communicating Sequential Processes (CSP) [HOA 78].

For the definition of data abstractions (i.e. values, data structures, and operations on them) LOTOS bases itself on the Abstract Data Type formalism of ACT ONE [EHR 85]. CCS and ACT ONE could then be considered the two parts that constitute LOTOS: the first for the "control" part, and the second for the "data" part.

The structure of a typical LOTOS specification could be graphically represented as a tree made up of many small, nested processes. LOTOS very much favours stepwise decomposition of specifications. For example, a service specification may, at the top level, consist of four processes: a sending entity, a receiving entity, and two bidirectional queues. These processes may themselves be subdivided, and this continues down to the point where we find very small processes such as the service primitives themselves. Some of these processes may share "gates", through which communication occurs. Also various control relationships may hold between processes. One process may be alternative to another, or may be in parallel with another, or may be capable of "disabling" another. All this is described in greater detail below.

It is not the purpose of this thesis to present a tutorial of LOTOS. However, since LOTOS is a relatively new language, it seems appropriate to present a brief overview of its main concepts. Another good reason for this brief overview, is showing the user how the two basic concepts of LOTOS are put together to write specifications, and help the reader in understanding the process of translating the data types texts existing in a LOTOS specification, as described in Chapter 4 (see §4.3). For a more extensive tutorial on LOTOS, see [BRI 86].

The remainder of this chapter is organized as follows: §2.2 describes the main concepts of the LOTOS "control" part. §2.3 describes the concepts of the LOTOS "data" part. §2.4 shows by means of examples how the two parts of LOTOS can be combined to write specifications. The syntax rules of LOTOS are not given here. However throughout this chapter LOTOS keywords are written in **boldface**.

2.2 LOTOS Control Concepts

In LOTOS, interprocess communication occurs by means of a two-way "rendezvous" mechanism, called "interaction". In an interaction, each one of the participating processes specifies the name of a "gate", and whatever information it wishes to provide on the values to be established. This can range from a specific value, to a type only. Processes participate in an interaction only if they specify the same gate and the information they provide is compatible. Thus, the concept of "information flow" strictly does not exist in LOTOS, but those who wish to retain it may still think that information flows from the processes that have "more" information on a value to those that have "less". For example,

$$g \quad ?x : int \quad !(3+5) \quad true$$

is an "action denotation" in LOTOS. It describes an 'interaction offer' at gate g, i.e. it means that this process is ready to interact with other processes in the

environment, which are also ready to execute ("matching") interaction offers. If it occurs, the interaction establishes a vector of three values: the first of these values, x, is not known to this process, hence the question mark. Only its sort (i.e. its type) is known to be "integer". The next two values instead are known, hence the exclamation mark. The first is the integer 8, and the second is the boolean "true". A "matching" offer could be something such as

$$g \quad !3 \quad ?y : int \quad ?z : bool$$

If a process becomes ready to execute this offer at the same time that another process is ready to execute the offer above, an interaction may occur, and then we can say that the process containing this second offer passes to the first the value 3 (which becomes the value of x in the first process), and receives the values 8 and "true", which become the values of y and z. After the interaction, each process will proceed independently. Another matching offer for the first one would be

$$g \quad ?y : int \quad ?z : int \quad !true$$

In this case, we see that the types of x and y match, but neither offer provides a value. An interaction can still occur, as it is assumed that "some" integer value will be established by the interaction mechanism, and sent to both processes. It is also interesting to note the way the two offers match in their third component, where they both agree on the value "true". In this case, there is no exchange or generation of values, but just synchronization on a value known by both.

An example of "non-matching" offer would be

$$g \quad ?3 \quad ?y : bool \quad ?z : bool$$

because the processes cannot agree on whether the second value should be boolean or integer.

A process that executes an offer will wait for a matching offer to occur. If the latter does not occur, the process containing the offer is said to be in "deadlock". No queuing is involved.

Action denotations are the basic building blocks of LOTOS. These building blocks can be composed in increasingly larger blocks (called "behavior expressions") by means of certain operators:

1. The operator ";" is used to prefix an action denotation to a behavior expression (first perform an offer, then proceed to the following behavior expression).

2. The operator "[]" is used to express alternatives. $B_1[]B_2$, where B_1 and B_2 are behavior expressions, then means: do either B_1 or B_2. The choice may be specified to be:

   * Completely "nondeterministic" (this is useful for example to express "internal events" or "spontaneous transitions");

   * Determined by the environment. In this case, each choice starts by an action denotation, and the choice is determined by what complementary offers are provided by the environment;

   * Determined by a "guard", i.e. a condition. This corresponds to the well-known "if" construct of programming languages. The notation for guards is: "$[\langle condition \rangle] \rightarrow B$", where B is a behavior expression.

3. The operator "||" is used to express parallel composition. $B_1 \parallel B_2$, where B_1 and B_2 are behavior expressions, then means: do in parallel B_1 and B_2. Furthermore, B_1 and B_2 may interact, by the mechanism described above, through the gates they share.

4. The operator ">>" is used to perform the sequential composition of behavior expressions ("enabling"). $B_1 >> B_2$, where B_1 and B_2 are behavior expression,

thus means: do B_1, then B_2. There is a mechanism by which B_1 can pass some values to B_2.

5. The operator "[>" is called "disabling" and is used to represent situations, very common in distributed systems, where a process is allowed to nondeterministically "interrupt" another process. The intuitive meaning of the expression B_1 [> B_2, where B_1 and B_2 are behavior expressions, is that process B_2 may disrupt or prevent the execution of process B_1. Whether B_2 is executed or not, it is a nondeterministic choice. Once the excution of B_1 has been disrupted by the start of B_2, B_1 will not resumed again, and B_2 will run to completion. If B_2 is not executed, B_1 completes its execution and the whole construct terminates.

6. Restriction or "hiding". As mentioned above, a LOTOS system can be seen as a "black box", say A, communicating with the external environment through gates. A itself may contain other black boxes, say B and C, which also communicate through gates. Some of the gates of B and C may also be gates of A. Others instead may be only for internal communication between B and C. These latter gates must be "hidden" inside A, i.e. not available for communication by the external word with A. This "hiding" is accomplished by the operator "\". Hence, $A \setminus [X]$, where A is a behavior expression, and X is a list of gate names, expresses the fact that gates in the list X are invisible outside A, i.e. can only be used for internal communication in A (note the value of this concept of "hiding" for writing modular specifications).

Furthermore, LOTOS has mechanisms to "name" behavior expressions (similar to procedure declarations in conventional programming languages), and mechanisms to "instantiate" behavior expressions, similar to procedure calls in conventional programming languages. "Named" behavior expressions are called "process

abstractions". Particularly noteworthy is the "gate parameterization" mechanism, by which different formal gates can play different actual roles in different instantiations. Gate parameters are shown between square brackets after the name of a process abstraction.

Interaction, enabling and instantiation are the three main ways of assigning a value to a LOTOS variable (a fourth one, the "let", is used to initialize a variable). In particular, LOTOS does not have the assignment statement or global variables.

2.3 LOTOS Data Type Concepts

Data definitions can occur anywhere in a specification written in LOTOS. Some data definitions are common to a large set of specifications, so they can belong to a library, other definitions are refinements of definitions introduced previously, ect. Data type definition in LOTOS is based on ACT ONE [EHR 85].

ACT ONE has the following features :

1. Modularization of specifications, which allows reference to already existing specifications in a library.

2. Combination of specifications.

3. Renaming of specifications.

4. Parameterization of specifications.

5. Actualization of parameterized specifications.

In this section, we introduce these concepts by means of examples. First, we define the concepts of signature, terms, equations, and conditional equations. Then, we will introduce the concepts of combination, parameterization, actualization, and renaming.

2.3.1 The Signature

In order to define an abstract data type one needs the possibility to define names of data carriers and operations. Such a definition must also include the domains and ranges of the operations. In ACT ONE, the names of the data carriers are called *sorts*. The definition of these sorts and operations is called *signature*. It only defines the syntax of a data type. Expressions can be formed either of sorts and operations or of a combination of both. Later in the examples sorts and operations are preceded by the keywords **sorts** (respectively **opns**).

2.3.2 Terms, Equations and Conditional Equations

The names of the sorts and operations defined in a signature must correspond to the data carriers and operations of a distinguished algebra, which is the semantics of the specification. All the elements of the data carriers can be produced by repeated calls to the operation symbols. The result of such a combination of operation symbol calls is called a *term*. Terms represent elements of a given sort, say s. Such terms are said to be of sort s and are referred to as s-term. For example if we have a specification of natural numbers represented by the sort *nat*, a constant $zero : -> nat$, and a unary operation $succ : nat -> nat$ then we can produce the following *nat*-terms:

$$zero, succ(zero), succ(succ(zero)), \cdots .$$

Each of these *nat*-terms can be interpreted as one element of the algebra of the natural numbers.

With the concept of signature all elements of an algebra can be represented. But what happens if we like to calculate with our natural numbers? We then have to introduce additional operation symbols e.g. the *plus* operator: $plus : nat, nat -> nat$.

With additional operations additional terms can be produced, e.g. $plus(zero, succ(succ(zero)))$. However because we do not have an understanding of the *plus* operator, we are not able to interpret the new *nat*-terms. So we need a new construct to express properties of operations. This construct is the equation. An equation is composed of two parts: a left-hand side and a right-hand side separated by the equality sign '='.

We write $succ^n(zero)$ to denote n subsequent calls to operation *succ*. We then define the intended semantics of the plus operation as follows:

$$plus(succ^n(zero), succ^m(zero)) = succ^{sum(n,m)}(zero).$$

Note that the result of the addition is represented by the sum of the calls of both summands.

Using the formalism introduced so far, in order to express all properties of the natural numbers, we would have to write infinitely many equations, since we can only add constant values i.e. $plus(3,5) = 8$. Thus we need to introduce variables to define equations, which are valid for a whole domain. The equation $plus(x,3) = succ^3(x)$ should not conflict with the natural numbers and indeed it does not. The problem of expressing the addition of the constant '3' to each other natural number can now be resolved by calling 3 times the operation *succ*. The variable x in *nat*-terms $succ(succ(succ(x)))$ can be considered as a null-ary operation i.e. is a constant like *zero*.

Note that the denotations of '3', '5' and '8' are not part of the syntactic definition, instead they are values i.e. interpretations of terms of a distinct sort e.g. $succ(succ(succ(zero)))$, $succ(succ(succ(succ(succ(zero)))))$ etc ....

Sometimes an equation is only applicable to a certain values in a specific domain, e.g. the division by zero is not allowed in the case of natural numbers. Thus

the need for a new construct to restrict the domain of applicability of operations. This construct is the conditional equation. A conditional equation is composed of two parts, the premises and the conclusion. The premises are conditions which restrict the domain of usage of the conclusion. The conclusion is an equation, and is applicable only if associated premises are verified. The premises and the conclusion in a conditional equation are separated by the implication sign '=>'.

With the introduction of terms with variables and the enlargement of the concept of signatures with equations and conditional equations, we now have the tools to specify algebras. Note that the equality sign used in the equations does not mean the equality of the terms on the left-hand side and right-hand side, but the equality of the interpretation of both terms if we replace the variables by actual values of the sorts of the variables.

The informal introduction of the *plus* operator can now be formalized by using the concept of *equation*. The new operator must be consistent with the already available operators *zero* and *succ*.

eqns forall x, y : *nat*
ofsort *nat*
$plus(x, zero) = x$;
$plus(x, succ(y)) = succ(plus(x, y))$;

The first equation expresses the semantics of the plus operator, when combined with the constant *zero*. Also the *succ* operator which was introduced to have access to the natural numbers with the exception of *zero*, is consistent with *plus* operator, since the result of its application can be represented by the *succ* operator. This is semantically correct according to the semantic definition of the plus operation given on the previous page.

2.3.3 Combination

Up to now it is difficult if not impossible to define systems with a large number of operations, equations, and conditional equations, since the mechanisms introduced so far do not provide concepts for the stepwise refinement and structuring of the specification of large systems. Hence one needs language concepts to combine operators, equations, and conditional equations belonging together, and to extend parts of a specification with additional operations. For these purposes ACT ONE provides the concept of *combination*. This concept is based on the fact that large specifications can split into smaller parts, while simple specifications may be combined (stepwise) into voluminous ones.

An example of combination is as follows : suppose that, one first has defined a complete data type definition called *nat_num* given by :

```
type nat_num is
    sorts nat
    opns zero : -> nat
         succ : nat -> nat
endtype
```

This type provides access only to the infinite set of natural numbers. One cannot use this definition to add two natural numbers. So one needs to enrich the above data type definition by the *plus*-operator. Enrichment of data type definitions is a special case of the combination, where no new sorts are added to the already defined data type. As a result of the enrichment of the *nat_num* specification, we obtain a new data type that we called *enriched_nat* given by:

```
type enriched_nat is nat_num
    opns plus : nat, nat -> nat
    eqns forall x, y : nat
         ofsort nat
         plus(x, zero) = x ;
         plus(x, succ(y)) = succ(plus(x, y)) ;
endtype
```

Note that equations are sorted by the sort of the operator they define. So in this example the two equations for *plus* are said to be of sort *nat*.

2.3.4 Parameterization and Actualization

There are several examples of abstract data types, such as strings, stacks, and queues, which are used in slightly different versions in a number of different specifications.

Speaking of queues we may have in mind queues of natural numbers, or queues of characters, queues of integers or queues of other abstract data type (for example, queues of queues).

Of course we could give separate specifications *queue*(*characters*), *queue*(*nat*), *queue*(*int*), or *queue*(*spec*) for each one of these abstract data types. But it would be much simpler to give only one specification for queues, which is in some sense parameterized, such that all these variants of queue specifications can be obtained by suitable actualization of the parameter.

For this purpose ACT ONE provides the concepts of *parameterization* and *actualization*. There are two requirements that are to be satisfied by the actual type:

1 The actual data type definition must have at least the same properties as the formal parameter.

2 The body of the actual parameter must be disjoint from the body of the data type definition that is to be actualized.

The second requirement induces that a data type definition cannot be actualized by itself, to eliminate this difficulty the data type is renamed prior to such actualization (see §2.3.6 for an example that involves this case). The following example may illustrate these techniques: suppose that a LOTOS specification deals

with two kinds of queues, e.g. queues of natural numbers and queues of characters. All we need is to define a parameterized data type *queue*(*data*) where *data* is the formal parameter, which can be actualized by the *nat_num* or *characters* data types definitions. The definitions of *queue*(*data*) is a follows:

```
type queue(data) is data with
    sorts queue
    opns new : − > queue
         rest : queue− > queue
         add : data, queue− > queue
         first : queue− > data
    eqns forall x : data, q : queue
         ofsort queue
           rest(new) = new ;
           ...
         ofsort data
           first(new) = d0 ;
           ...
endtype
```

where *data* is a formal data type definition such as:

```
type data is
    formalsorts data
    formalopns d0 : − > data
endtype
```

Given the parameterized definition we can now instantiate *data* by *nat_num* to obtain a definition of *nat_num_queue* as shown below:

```
type nat_num_queue is
  queue(data) actualizedby nat_num using
  sortnames nat for data
  opnames zero for d0
endtype
```

Replacing *nat_num* above by some other formal type *characters* we obtain a definition for the queue of characters given by:

```
type characters_queue is
  queue(data) actualizedby characters using
  sortnames char for data
  opnames e for d0
endtype
```

2.3.5 Renaming

Renamings allow to change names of sorts and equation symbols in a given specifications without changing their semantics. This concept is useful during the development of a specification in the case where an already defined abstract data type is needed in a specific environment. Renamings could of course be done explicitly by rewriting the data type definition with new sorts and operations. This induces a renaming of the declaration of the operation symbols, and the declaration of variables of terms, equations, and conditional equations. Because the semantics of the data definition to be renamed have to be maintained, this would be a clumsy way to operate.

The concept of *renaming* allows to avoid this rewriting. Here is an example that illustrates this technique: assume that the data type definition *queue*(*data*) of the last section should be used in the OSI transport service environment which deals with connection objects and with data objects to be transfered. Then the renamed *queue*(*data*) could be written as follows:

```
type connect(objects) is
  queue(data) renamedby
    sortnames channel for queue
              objects for data
    opnames   send for add
              receive for first
endtype
```

2.3.6 A General Example

After a brief introduction to the LOTOS data concepts, we give now an example that uses all the concepts introduced so far. This example consists of a definition of the set of element *set(element)*, where *element* is a formal parameter that can be actualized by another data type to obtain a set of objects defined by the actual type. Several operations can be defined for sets: the membership operation which returns *true* or *false* depending of whether an element belongs to the set or not. Hence the set definition requires the existence of the *boolean* definition. The cardinality of a set is another case of operation, which gives the size of a set. Hence the need of the *natural_number* definition. Let us suppose that the definitions for *boolean* and *nat_nums* exist somewhere in the library. Then the definition of *set(element)* simply imports them by means of the combination mechanism described in §2.3.3. Before we give the definition of *set(element)*, we first give the definition of its formal parameter *element*. This definition is given in Figure 2.1. The operators between two periods are infix binary operator.

```
type element is boolean with
    formalsorts element
    formalopns .eq., .ne., : element, element → bool
    formaleqns forall x, y, z : element
                   ofsort bool
                   x eq x = true ;
                   x eq y = y eq x ;
                   x eq y, y eq z ⇒ x eq z = true ;
                   x ne y = not(x eq y) ;
    endtype
```

Figure 2.1 The *element* ADT Definition

Note that the operation *not* is the one of the *boolean* definition which is imported by the data type *element*. The formal equations in the definition of *element* introduce the formal requirements on boolean equality and inequality i.e., these requirements must be satisfied by the actual type (see §2.3.4).

Now, we are ready to give the definition of *set*(*element*). This definition is given in Figure 2.2. It is characterized among other things by two operators *insert* and *remove*. *Insert* adds an element to the set if this element does not belong to it, the semantic definition of this operator is given by equations 1 to 3 in Figure 2.2. The *remove* operator removes an element from the set if this element belong to it, the semantic definition of this operator is given by equation 4 and 5 in Figure 2.2 (p. 23).

At this point, one could be interested in defining the set of set *set*(*set*) data type. The first idea that comes to mind is to actualize the definition of *set*(*element*) by itself which is not correct (see requirement 2 in §2.3.4). So this can be accomplished by first renaming the definition of *set*(*element*) to obtain, say *set*1(*element*1), then actualize *set*(*element*) by *set*1(*element*1) to obtain another data type *set*(*set*1), and finally rename this latter definition to obtain *set*(*set*). This sequence of definitions is given in Figure 2.3

---

type *set*1(*element*1) **is** *set*(*element*) **renamedby**
    sortnames *set*1 **for** *set*
        *element*1 **for** *element*
endtype

type *set*(*set*1) **is** *set*(*element*) **actualizedby** *set*1(*element*1) **using**
    sortnames *setofset* **for** *set*
        *set*1 **for** *element*
endtype

type *set*(*set*) **is** *set*(*set*1) **renamedby**
    sortnames *set* **for** *set*1
        element **for** *element*1
endtype

Figure 2.3 Steps to Define the *set*(*set*) ADT Definition

```
type set(element) is element with boolean with nat_nums with
        sorts set
        opns {} : → set
             insert, remove : element, set → set
             .isin., .notin. : element, set → bool
             .union., .ints., .minus. : set, set → set
             .eq., .ne., .includes. : set, set → bool
             card : set → nat
        eqns forall x, y : element, s, t, u : set
             ofsort set
             insert(x, insert(x, s)) = insert(x, s) ;
             insert(x, insert(y, s)) = insert(y, insert(x, s)) ;
             x notin s ⇒ remove(x, insert(x, s)) = s ;
             x notin s ⇒ remove(x, s) = s ;

             {} union s = s ;
             s union t = t union s ;
             insert(x, s) union t = insert(x, s union t) ;

             {} ints s = {} ;
             s ints t = t ints s ;
             x isin t ⇒ insert(x, s) ints t = insert(x, s ints t) ;
             x notin t ⇒ insert(x, s) ints t = s ints t ;

             s minus {} = s ;
             s minus t = s minus (s ints t) ;
             s minus insert(x, t) = remove(x, s) minus remove(x, t) ;

             ofsort bool
             x isin {} = false ;
             x isin insert(y, s) = (x eq y) or (x isin s) ;
             x notin s = not(x isin s) ;

             t minus s = {} ⇒ s includes t = true ;
             t minus s = insert(x, u) ⇒ s includes t = false ;

             s eq t = (s includes t) and (t includes s) ;
             s ne t = not(s eq t) ;

             ofsort nat
             card({}) = 0 ;
             x notin s ⇒ card(insert(x, s)) = succ(card(s)) ;
endtype
```

Figure 2.2 The *set(element)* ADT Definition

2.4 Putting Together Data and Control to Write Specifications — Examples

In this section, we demonstrate the use of ADTs in LOTOS specifications by means of examples. The first example, due to Ed Brinksma, shows the LOTOS definition of an "unbounded queue" defined recursively as a queue of processes, each capable of handling a data item of sort *nat*. At each moment in time, this queue can be visualized as being made up of a subqueue and a *first* process, which holds the value of the first element in the queue. *First* offers its contents to the environment. Offering a data item to *uqueue* causes the creation of a new process within *uqueue*, which will be able to propagate its contents to *first* via a chain of *mid* gates. The only data type definition needed in this example is *nat* for natural number. The code for this "unbounded queue" definition is shown in Figure 2.4.

```
specif unboundq

    type nat is
        sorts nat
    endtype

  process uqueue[ingate, outgate] :=
        ingate?X : nat ;
        ( uqueueingate, outgate || first[mid, outgate] (X) ) \ mid
            where
                process first[ingate, outgate] (X : nat) :=
                    outgate!X ; ingate?Y : nat ; first[ingate, outgate] (Y)
                endproc
  endproc
endspec
```

Figure 2.4 An Unbounded Queue of Processes

In this example we have defined the "unbounded queue" as a queue of processes. However this definition can be rewritten using a specification of queue mechanisms in terms of ADTs, rather than processes. Figure 2.5 gives the code for this speci-

fication (this second example is also due to Ed Brinksma, who, in an unpublished paper, was able to prove the equivalence of these two specifications).

In this respect, it is interesting to note that, opposite to comparable languages, LOTOS has no primitive for queues. Rather, queues must be explicitly defined when needed, possibly by including one of the queues provided by the library. In this way, any desired queue may be introduced: not only conventional FIFO queues, but also queues with priority, passing, erasure, etc. For more complex examples (see [SCO 85]).

```
specif adtqueue
    type nat1 is boolean1 with boolean with
        sorts nat
        opns if_then_else : bool, nat, nat → nat
        eqns forall M, N : nat in
            if_then_else(true, M, N) = M ;
            if_then_else(false, M, N) = N ;
    endtype
    type boolean1 is boolean with
        opns true, false : → bool
             not : bool → bool
        eqns
            not(true) = false ;
            not(false) = true ;
    endtype
    type buffer is nat1 with boolean1 with boolean with
        sorts queue
        opns new : → queue
             error : → nat
             add : nat, queue → queue
             rem : queue → queue
             first : queue → queue
             empty : queue → bool
             if_then_else : bool, queue, queue → queue
        eqns forall M : nat, Q, Q1, Q2 : queue in
        ofsort queue
            rem(new) = new ;
            rem(add(M, Q)) = if_then_else(empty(Q), new,
                    add(M, rem(Q))) ;
            if_then_else(true, Q1, Q2) = Q1 ;
            if_then_else(false, Q1, Q2) = Q2 ;
        ofsort nat
            first(new) = error ;
            first(add(M, Q)) = if_then_else(empty(Q), M,
                    first(Q)) ;

        ofsort bool
            empty(new) = true ;
            empty(add(M, Q)) = false ;
    endtype
      process fifo[ingate, outgate] (S : queue) :=
        ingate?X : nat ; fifo[ingate, outgate] (add(X, Q))
        []
        [ not(empty(S)) ] →
        ( outgate!first(S) ; fifo[ingate, outgate] (rem(S)) )
      endproc
endspec
```

Figure 2.5 An Unbounded Queue Described by an ADT Definition

CHAPTER 3

Theories of Term Rewriting Systems and Conditional Term Rewriting Systems

3.1 Introduction

This chapter presents the theories of term rewriting systems and conditional term rewriting systems. We start by some definitions related to terms and substitutions. We then discuss equational and conditional theories. Term rewriting systems are described together with the process of rewriting. Two properties of rewriting systems, termination and confluence, are defined, and shown to provide a decision procedure for equational theories. We show how the Knuth-Bendix completion procedure can be used to generate such a decision procedure. We then describe conditional rewriting systems, the process of rewriting with conditional rules, we address the problems of one step and global termination for conditional rewriting systems, and we present a sufficient condition to deal with these two problems. A term rewriting system that satisfies this condition is said to be *fair*. Then we discuss the problem of confluence for fair conditional rewriting systems, and show how the Knuth-Bendix completion procedure can be extended to fair conditional rewriting systems. Finally we present methods used for constructing terminating rewriting systems, particularly the recursive path ordering (RPO) mechanism. Our developement here takes an operational view of rewriting. See [HUE 80a] and [KAP 84b] for a treatment using relations.

3.2 Terms and Substitutions

Let X be a denumerable set of symbols called *variables*, denoted x, y, z. Let F be a finite set disjoint from X, graded by an *arity function* $a: F \rightarrow N$. Elements in F are called *operators symbols*. Examples of operators are *cons* in lists, $+$ in arithmetics, and *true* in boolean. Let T be the *set of terms*. We define $F_n = \{f \in F \mid a(f) = n\}$.

A *term* is defined as either (a) a variable, or (b) being of the form $f(t_1, \ldots, t_n)$, where f is an operator in F_n, called the *root operator*, and $t_1, \ldots, t_n$ are called the *arguments* of f. An operator with zero arity is called a *constant*.

We now define a few functions on terms:
$\mathcal{V}(t) \subset X$ (the set of variables of t) such that:

a) $\mathcal{V}(x) = \{x\}\ \forall x \in X$.

b) $\mathcal{V}(f(t_1, \ldots, t_n)) = \cup_{i=1}^{n} \mathcal{V}(t_i) \forall f \in F_n$.

When $\mathcal{V}(t) = \emptyset$, then t is said to be a *ground term*.

The *subterms* of a term are the term itself and the subterms of its arguments. We now formalize the notion of occurrence of a subterm in a term. Let N_+^* be the set of finite sequences of positive integers. We use Λ and $\cdot$ to denote the empty sequence in N_+^* and the concatenation operation on sequences. We shall call the members of N_+^* *occurrences* and denote them u, v, w. A subterm s within a term can be designated by an occurrence which is a member of N_+^*. For any term $t \in T$, we define its *set of occurrences* $\mathcal{O}(t) \subseteq N_+^*$ as follows:

a) $\mathcal{O}(t) = \{\Lambda\}$ if t is a variable or constant.

b) $\mathcal{O}(t) = \{\Lambda\} \cup \{i \cdot u \mid i \leq n, u \in \mathcal{O}(t_i)\}$ if $t = f(t_1, \ldots, t_n)$.

For example, if $t_1 = f(x, g(x, h(y)))$ then $\mathcal{O}(t) = \{\Lambda, 1, 2, 2 \cdot 1, 2 \cdot 2, 2 \cdot 2 \cdot 1\}$.

For $u \in \mathcal{O}(t)$, t/u is the subterm of the term t at occurence u, defined by:

a) $t/\Lambda = t$.

b) $t/i \cdot u = t_i/u$ if $t = f(t_1, \ldots, t_n)$.

For example $t_1/2 \cdot 1 = x$.

For $t \in T, u \in O(t)$ and $s \in T$, we define recursively $t[u \leftarrow s]$ (to denote the term t with the subterm at occurrence u replaced by term s) by:

a) $t[\Lambda \leftarrow s] = s$.

b) $f(t_1, \ldots, t_n)[i \cdot u \leftarrow s] = f(t_1, \ldots, t_{i-1}, t_i[u \leftarrow s], t_{i+1}, \ldots, t_n),\ i \leq n$.

A *substitution* is a mapping σ from X to T, with $\sigma(x) = x$ for all but a finite number of variables. The domain of the substitution is extended to the set of all terms T by defining:

$$\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n)).$$

A substitution σ is represented by a finite set of ordered pairs, denoted $\sigma = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$. For example, consider the term $t_1 = f(g(x), h(z), y)$ and the substitution $\sigma = \{x \leftarrow h(y), z \leftarrow g(g(x)), y \leftarrow x\}$. We can apply σ to obtain $\sigma(t_1) = f(g(h(y)), h(g(g(x))), x)$.

A *unifying substitution* or *unifier* of two terms is a substitution which when applied to the two terms results in an identical expression. Formally two terms s and t, are said to be *unifiable* if and only if there exist a substitution, σ, such that $\sigma(s) = \sigma(t)$. For example, if $s = f(g(y), z)$ and $t = f(x, h(y))$, one unifier for s and t can be $\sigma_1 = \{x \leftarrow g(f(x_1)), y \leftarrow f(x_1), z \leftarrow h(f(x_1))\}$. For this unifier, $\sigma_1(s) = \sigma_1(t) = f(g(f(x_1)), h(f(x_1)))$. Usually there is more than one unifier (if it exists) for two terms, one of these unifiers is called the *most general unifier* (*mgu* for short), and every other unifier contains *mgu* as a factor (in terms of functional decomposition.) The most general unifier is unique, up to variables renaming. For

the previous example, $mgu = \{x \leftarrow g(y), z \leftarrow h(y)\}$, and σ_1 can be expressed as a functional composition $\sigma_1 = \sigma_2 \circ mgu$, where $\sigma_2 = \{y \leftarrow f(x_1)\}$. The *unification* of two terms, s and t, is $mgu(s)$ (which is the same as $mgu(t)$ for their most general unifier, mgu). For s and t above, $f(g(y), h(y))$ is their unification.

Unification is one of the basic concepts used in computational logic, and plays a central role in the inference rules of resolution [ROB 65] and logic programming [KOW 74]. We shall use it in the superposition algorithm in § 3.5. Algorithms to compute the most general unifier have been proposed by several authors (among others [ROB 71],[BAX 73],and [HUE 78]). All these algorithms are non-linear; a linear algorithm is given in [PAT 78]. Two efficient algorithms are described in [MAR 82] and [COR 83].

One-way unification (when unificatio is permitted in only one of the terms) is called *matching.* We say that a term s *matches* a term t (or s is an *instance* of t, or t is a *generalization* of s), if and only if there exists a substitution σ such that $s = \sigma(t)$. When the domain of σ is restricted to $\mathcal{V}(t)$, σ is unique and is called the *match* of s by t. For example, $s = f(h(x), g(h(x)))$ matches with $t = f(y, g(y))$, and the match of s by t is $\sigma = \{y \leftarrow h(x)\}$.

3.3 Equational and Conditional Theories

3.3.1 Equational Theories

An *equation* is a pair $s = t$ where s and t are terms. In equations , all variables are (implicitly) universally quantified. A *ground instance* of an equations $s = t$, is an equation $\sigma(s) = \sigma(t)$, that contains no variables, where σ is some substitution.

The equality $=_{\mathcal{E}}$ generated by a set $\mathcal{E}$ of equations , contains all pairs $\sigma(s) = \sigma(t)$ for $s = t$ in $\mathcal{E}$ and σ an arbitrary substitution. In equational logic $=_{\mathcal{E}}$ is called an

equational theory, and $\mathcal{E}$ a *base* or a *set of axioms* for the theory. The *equational theory* of $\mathcal{E}$ consists of the closure of $\mathcal{E}$ under the following rule of inference: reflexivity, symmetry, transitivity, universal instantiation, and replacement of equals by equals. If an equation, $s = t$, is in $=_{\mathcal{E}}$, we say that $s = t$ is an *equational theorem* (or *equational consequence*) of $\mathcal{E}$, that we write $s =_{\mathcal{E}} t$. Now we will see by means of example how the rules of inference are used to formally prove that a certain equation is an equational theorem of a set of axioms: let

$$\mathcal{E} = \{+(0,x) = x,\ +(i(x),x) = 0,\ +(+(x,y),z) = +(x,+(y,z))\},$$

(note that $\mathcal{E}$ is the set of axioms for groups, if we take $+$ to be the group's binary operation, $i(x)$ to be the inverse of x and 0 to be the identity). Given this set of axioms $\mathcal{E}$, we want to use the rules of inference to prove that:

$$(E1) \qquad i(i(x)) = x$$

is an equational theorem. Note that each proof step is itself an equational theorem. The proof of this equation is as follows:

Given:

(1) $+(0,x) = x$

(2) $+(i(x),x) = 0$

(3) $+(+(x,y),z) = +(x,+(y,z))$

Apply $\sigma = \{x \leftarrow i(x)\}$ to (2) gives:

(4) $+(i(i(x)),i(x)) = 0$

Insert (4) into (1) gives:

(5) $+(+(i(i(x)),i(x)),x) = x$

apply $\sigma = \{x \leftarrow i(i(x)), y \leftarrow i(x), z \leftarrow x\}$ to (3) gives:

(6) $+(+(i(i(x)),i(x)),x) = +(i(i(x)),+(i(x),x))$

insert (2) into (6) gives:

(7) $+(+(i(i(x)),i(x)),x) = +(i(i(x)),0)$

by (7) and (5) and transitivity, we get:

(8) $+(i(i(x)),0) = x$

apply $\sigma = \{x \leftarrow 0\}$ to (1) gives:

(9) $+(0,0) = 0$

insert (9) into (8) gives:

(10) $+(i(i(x)),+(0,0)) = x$

apply $\sigma = \{x \leftarrow i(i(x)), y \leftarrow 0, z \leftarrow 0\}$ to (3) gives:

(11) $+(+(i(i(x)),0),0) = +(i(i(x)),+(0,0))$

by (11) and (10) and transitivity, we get:

(12) $+(+(i(i(x)),0),0) = x$

insert (2) into (12) gives:

(13) $+(+(i(i(x)),+(i(x),x)),0) = x$

insert (6) into (13) gives:

(14) $+(+(+(i(i(x)),i(x)),x),0) = x$

insert (4) into (14) gives:

(15) $+(+(0,x),0) = x$

insert (1) into (15) gives:

(16) $+(x,0) = x$

apply $\sigma = \{x \leftarrow i(i(x))\}$ to (16) gives:

(17) $+(i(i(x)),0) = i(i(x))$

by (8) and (17), transitivity and symmetry, we get:

(18) $i(i(x)) = x$

Note that the group axioms $\mathcal{E}$, as given above, state that 0 is the right identity for x, but not that it is the left identity. However, during the proof steps for equation $(E1)$, we show at step (16) that $+(x,0) = x$ is an equational consequence of the axioms. Lemmas such as this one can be produced by the automatic theorem proving method based on the Knuth-Bendix completion procedure discussed in §3.5.

Most equational proofs are tedious and time consuming to construct by hand, hence the search for automated proof methods.

3.3.2 Conditional Theories

A *conditional equation* is a formula of the form: $\wedge_{i=1}^{n} p_i = q_i \Rightarrow s = t$, where p_i, q_i, s and t are terms. In conditional equations, all variables are (implicitly) universally quantified. The operational semantics of a conditional equation is that the equation on the right-hand side of the implication sign is applicable only if the conjunction of conditions at the left-hand side is verified. When $n = 0$ we simply have an equation. A *ground instance* of conditional equation $\wedge_{i=1}^{n} p_i = q_i \Rightarrow s = t$, is a conditional equation $\wedge_{i=1}^{n} \sigma(p_i) = \sigma(q_i) \Rightarrow \sigma(s) = \sigma(t)$, that contains no variables, where σ is some substitution.

The equality $=_{\mathcal{E}_c}$ generated by a set $\mathcal{E}_c$ of conditional equations, contains all formulas $\wedge_{i=1}^{n} \sigma(p_i) = \sigma(q_i) \Rightarrow \sigma(s) = \sigma(t)$ for $\wedge_{i=1}^{n} p_i = q_i \Rightarrow s = t$ in $\mathcal{E}_c$ and σ an arbitrary substitution. We call $=_{\mathcal{E}_c}$ a *conditional theory*, and $\mathcal{E}_c$ a *set of conditional axioms* for the theory. The *conditional theory* of $\mathcal{E}_c$ consists of the closure of $\mathcal{E}_c$ under the same rules of inference as in the equational case (see §3.3.1). The only difference is that if we replace a term in a conditional equation by another term of another equation to obtain a new conditional equation, then the conjunction of conditions of the latter is equal to the conjunction of the conditions of the two other conditional equations. If a conditional equation $\wedge_{i=1}^{n} p_i = q_i \Rightarrow s = t$ is in $=_{\mathcal{E}_c}$, we

say that $\wedge_{i=1}^{n} p_i = q_i \Rightarrow s = t$ is a *conditional theorem* of $\mathcal{E}_c$. Let us now see an example of the application of the rules of inference in the conditional case. Let

$\mathcal{E}_c = \{$

$eq(x, x) = true,$

$eq(x, y) = true \Rightarrow isin(x, insert(y, s)) = true,$

$isin(x, insert(y, s)) = true \Rightarrow insert(x, insert(y, s)) = insert(y, s)\}$

be the set of conditional axioms for sets, where *eq* is the equality operation, *isin* the membership operation, while *insert* is the set insertion operation. Note that this set of axioms is not complete in terms of the intuitive interpretation. The variables x, y are of sort *data*, and s is a variable of sort $set(data)$. Given this set of conditional axioms $\mathcal{E}_c$, we want to prove that:

$(E2)$ $isin(x, insert(y, s)) = true \wedge eq(z, x) = true \Rightarrow isin(z, insert(y, s)) = true$

is a conditional theorem. The proof of this conditional equation is as follows: Given:

(1) $eq(x, x) = true$

(2) $eq(x, y) = true \Rightarrow isin(x, insert(y, s)) = true$

(3) $isin(x, insert(y, s)) = true \Rightarrow insert(x, insert(y, s)) = insert(y, s)$

Apply $\sigma = \{x \leftarrow z\}$ to (2) gives:

(4) $eq(z, y) = true \Rightarrow isin(z, insert(y, s)) = true$

Apply $\sigma = \{y \leftarrow x, s \leftarrow insert(y, s)\}$ to (4) gives:

(5) $eq(z, x) = true \Rightarrow isin(z, insert(x, insert(y, s))) = true$

By (3) and (5) and transitivity, we get:

(6) $isin(x, insert(y, s)) = true \wedge eq(z, x) = true \Rightarrow isin(z, insert(y, s)) = true$

As for the equational case, theorems like this one can be produced by the automatic theorem proving method based on Knuth-Bendix-like completion procedure discussed in §3.6.4.

3.4 Term Rewriting Systems

In this section, we will develop one of the main paradigms of computing with equations, using them as rewrite rules over terms. Term rewriting systems are the basis for, on one hand, the proof of theorems in equational theory and, on the other hand, the construction of abstract interpreters for directed equations considered as programming languages.

A *rewrite rule* (or just a *rule*) is a directed pair of terms, written $\lambda \to \rho$ such that $\mathcal{V}(\rho) \subseteq \mathcal{V}(\lambda)$. This condition is necessary to avoid the resolution of equation in the theory represented by the term rewriting system during rewriting.

The fundamental difference between equations and rewrite rules is that equations denote equality, whereas rewrite rules are treated directionally, as one-way replacements. Furthermore, the only substitutions required for rewrite rules are the ones found by pattern matching. One may *reduce* (or *rewrite*) a term t using a rewrite rule $\lambda \to \rho$ if there is an occurence $u \in \mathcal{O}(t)$ such that t/u matches λ. The reduced (rewritten) form of t is $t[u \leftarrow \sigma(\rho)]$, where σ is the match of t/u by λ. For example, if we have $(\lambda \to \rho) = f(x, g(x,y)) \to k(x,y)$ and $t = f(z, f(a, g(a, h(z))))$, $t/2$ matches λ, $\sigma = \{x \leftarrow a, y \leftarrow h(z)\}$, and t is reduced to $f(z, k(a, h(z)))$.

A *Term Rewriting System* $\mathcal{R}$ (henceforth called TRS for short) is a finite set of rewrite rules. We write $s \to_{\mathcal{R}} t$ if and only if s can be reduced to t using one of the rewrite rules in $\mathcal{R}$ exactly once. From now on, we shall use $\to$ for $\to_{\mathcal{R}}$ when it is clear from the context. We write $s \to^* t$ to mean that t can be obtained from s by applying rules from $\mathcal{R}$ zero or more times. We write $t \leftrightarrow^* t'$, when there exist zero or more terms $s_1, \ldots, s_n$ such that $t \rightleftharpoons s_1 \rightleftharpoons, \ldots, \rightleftharpoons s_n \rightleftharpoons t'$, where $\rightleftharpoons$ denotes ($\leftarrow$ or $\to$). For example $\mathcal{R} = \{b \to a, f(b,x) \to g(x,x)\}$, we have $g(b,b) \leftrightarrow^* f(a,a)$, since $g(b,b) \to g(b,a) \to g(a,a) \leftarrow f(b,a) \to f(a,a)$.

Note that $\leftrightarrow^*$ is the same as the equational theory of a rewriting system $\mathcal{R}$, denoted $=_{\mathcal{R}}$, when $\mathcal{R}$ is considered as a set of equations. Conversely any set of equations $\mathcal{E}$ can be transformed into a rewriting system $\mathcal{R}$, using the following technique, suggested in [KNU 70] and [HUE 80a]: for every equation $s = t$ in $\mathcal{E}$, choose nondeterministically one of the following:

(1) If $\mathcal{V}(s) \subseteq \mathcal{V}(t)$, put $t \rightarrow s$ in $\mathcal{R}$.

(2) If $\mathcal{V}(t) \subseteq \mathcal{V}(s)$, put $s \rightarrow t$ in $\mathcal{R}$.

(3) If $\mathcal{V}(s) \cap \mathcal{V}(t) = \{x_1, \ldots, x_n\}$. Introduce a new operator f that does not appear in $\mathcal{E}$ or $\mathcal{R}$, and put the two rules $s \rightarrow f(x_1, \ldots, x_n)$ and $t \rightarrow f(x_1, \ldots, x_n)$. into $\mathcal{R}$.

The resulting rewriting system $\mathcal{R}$ will have the same equational theory as $\mathcal{E}$, except for the possible presence of new operators, due to rule (3). According to [KNU 70] and [HUE 80a] these rules enable one to transform a set of axioms $\mathcal{E}$, into a rewriting system $\mathcal{R}$, such that $s =_{\mathcal{E}} t$ if and only if $s =_{\mathcal{R}} t$ for all terms s and t. We know that $s =_{\mathcal{R}} t$ if and only if $s \leftrightarrow^* t$. Thus, if we have a decision procedure for $\leftrightarrow^*$, we have a decision procedure for the equational theory of $\mathcal{E}$. One can decide $\leftrightarrow^*$ if $\mathcal{R}$ has two properties, namely confluence and termination.

A rewriting system $\mathcal{R}$ is *Church-Rosser* if and only if, for all terms s and t, $s =_{\mathcal{R}} t$ if and only if there exists a term s' such that $s \rightarrow^* s'$ and $t \rightarrow^* s'$.

An equivalent characterization is "confluence". $\mathcal{R}$ is *confluent* if and only if for all terms s, t, and s', $s' \rightarrow^* s$ and $s' \rightarrow^* t$ implies there is some term s'' such that $s \rightarrow^* s''$ and $t \rightarrow^* s''$.

We say that a term t is *irreducible* or in *normal form* (relative to $\mathcal{R}$) if and only if there is no s such that $t \rightarrow s$; that is no subterm of t is an instance of some

left-hand side of a rule in $\mathcal{R}$. We say that t is a *$\mathcal{R}$-normal form* of s if and and only if $s \rightarrow^* t$ and t is a normal form relative to $\mathcal{R}$.

When a TRS $\mathcal{R}$ is confluent, the normal form of any term is unique, when the normal form exists. A sufficient condition for the existence of such canonical forms is the termination of all rewritings: we say that a TRS $\mathcal{R}$ is *noetherian* (*terminating, finitely terminating*) if and only if there is no term t_1 for which there exists an infinite chain of reductions $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \cdots$. If $\mathcal{R}$ terminates, any term t has at least one normal form, denoted $t\downarrow$. For example, the rewriting system $\mathcal{R} = \{f(x, f(y, z)) \rightarrow f(f(x, y), z)\}$ is noetherian. However, the rewriting system $\mathcal{R} = \{f(x, y) \rightarrow f(y, x)\}$ is not neotherian because we have $f(a, b) \rightarrow f(b, a) \rightarrow f(a, b) \rightarrow \cdots$.

Unfortunately, it is undecidable whether an arbitrary TRS terminates [HUE 78]. However, a number of methods have been proposed that prove termination in particular cases (see [KNU 70],[MAN 70],[LAN 75a],[LIP 77], [PLA 78a],[PLA 78b],[DER 79a],[DER 82],[GUT 83], and [JOU 82a]). The method used in our Knuth-Bendix implementation is the one presented in [DER 82], and described in §3.7. It uses a *reduction odering*, defined to be any well-founded partial ordering $\succ$, on terms, such that $s \succ t \Rightarrow f(\cdots, s, \cdots) \succ f(\cdots, t, \cdots)$ and $\sigma(s) \succ \sigma(t)$ for any terms s, t, $f(\cdots, s, \cdots)$, and $f(\cdots, t, \cdots)$ and any substitution σ [MAN 70]. The termination proof consists of showing that $\lambda \succ \rho$ for every rule $\lambda \rightarrow \rho$ in $\mathcal{R}$.

When $\mathcal{R}$ is both noetherian and confluent, it is said to be *complete* (or *convergent* or *canonical*). Thus, for a complete rewriting system $\mathcal{R}$, every term has a unique normal form. Furthermore, $\leftrightarrow^*$ and hence $=_{\mathcal{R}}$ is decidable when $\mathcal{R}$ is convergent: $(s =_{\mathcal{R}} t)$ if and only if $(s \leftrightarrow^* t)$ if and only if $(s\downarrow = t\downarrow)$. To test whether $s =_{\mathcal{R}} t$, one can reduce both terms to normal form (by applying arbitrary reductions) and then check whether the normal forms are identical. Since $\mathcal{R}$ terminates, this procedure

is effective; reductions cannot continue indefinitely. Confluence is undecidable for arbitrary TRS. However we will see now that confluence is decidable for noetherian systems.

A rewriting system $\mathcal{R}$ is *locally confluent* if and only if for all terms s, t, and s', $s' \rightarrow t$ and $s' \rightarrow s$ implies there is some s'' such that $t \rightarrow^* s''$ and $s \rightarrow^* s''$. The difference between the two definitions of confluence and local confluence is in the number of reductions permitted to obtain t and s from s'. Note that confluence implies local confluence. The converse is not necessarily true. For example, $\mathcal{R} = \{a \rightarrow b, a \rightarrow c, b \rightarrow a, b \rightarrow d\}$ is locally confluent, even though a has two distinct normal forms, namely c and d. However, it is proved in [NEW 42] that when $\mathcal{R}$ is noetherian the confluence is equivalent to local confluence. Here is the theorem.

Theorem 1. ***A noetherian TRS $\mathcal{R}$ is confluent if and only if it is locally confluent.***

Sevaral equivalent or related "diamond lemmas" have been shown in [KNU 70] and [HUE 80b]. The theorem above reduces the test of confluence to the one of local confluence. Now we will see how local confluence of term rewriting systems is decidable. The following definitions are needed.

Two terms are said to *overlap* if and only if one is unifiable with a non-variable subterm of the other, and the two terms share no variables. The superposition of two overlapping terms is the corresponding unification of one term and a subterm of the other term. To superpose two rewrite rules the following algorithm is used:

Superposition Algorithm:

Let $\lambda_1 \rightarrow \rho_1$ and $\lambda_2 \rightarrow \rho_2$ be two rules in a rewriting system $\mathcal{R}$, such that λ_1 and λ_2 overlap at occurence u of λ_1, and let σ be the *mgu* of λ_1/u and λ_2. (We assume that we have renamed variables appropriately so that λ_1 and λ_2 share

no variables.) We then say that the pair $\langle \sigma(\lambda_1[u \leftarrow \rho_2]), \sigma(\rho_1) \rangle$ is *critical* in $\mathcal{R}$. For example, consider the two rules $f(x, g(x, h(y))) \rightarrow k(x, y)$ and $g(a, z) \rightarrow l(z)$. We can superpose the first rule at occurence 2 with the the second one, using the *mgu* $\{x \leftarrow a, z \leftarrow h(y)\}$ obtaining the critical pair $\langle f(a, l(h(y))), k(a, y) \rangle$.

For a finite rewriting system $\mathcal{R}$, there are finitely many critical pairs. They can be effectively computed using the the standard unification algorithm. The utility of critical pairs is apparent in the following theorem.

Theorem 2. *A TRS $\mathcal{R}$ is locally confluent if and only if for every critical pair $\langle s, t \rangle$ of $\mathcal{R}$ we have $s \downarrow = t \downarrow$.*

The original version of this Theorem is presented in [KNU 70] where it is combined with Newman's Theorem (Theorem 1). The version of the theorem given above appears in [HUE 80], and does not require termination.

Combining Theorems 1 and 2 gives us a decision procedure for the confluence of terminating TRS's. When a TRS system $\mathcal{R}$ is noetherian but non-confluent, we can in some cases complete it to produce a complete (convergent) rewriting system having the same equational theory. This is the subject of the next section.

3.5 The Knuth-Bendix Completion Procedure (Huet's Version)

Consider a rewriting system $\mathcal{R}$, and a reduction ordering $\succ$, such that $\lambda \succ \rho$ for all rules $\lambda \rightarrow \rho$ in $\mathcal{R}$. Theorem 2 of last section indicates that to test for local confluence, one can check if for every critical pair $\langle s, t \rangle$ we have $s \downarrow = t \downarrow$. The two terms forming a critical pair are the result of rewriting a single term by two different rules in $\mathcal{R}$, after applying a substitution. Consequently, the equation $s = t$ formed by the critical pair is in $=_{\mathcal{R}}$, and the rule $s \downarrow \rightarrow t \downarrow$ or $t \downarrow \rightarrow s \downarrow$ may be added to $\mathcal{R}$ without changing $=_{\mathcal{R}}$. Furthermore, if the two sides of the added rule

are ordered under $\succ$ in the appropriate direction, the termination of $\mathcal{R}$ is preserved. Thus, if the local confluence test fails, (i.e. if we found a critical pair $\langle s, t \rangle$ such that $s \downarrow \neq t \downarrow$), and the normalized critical pair is orderable, we may add the rule formed from that orderable critical pair, and test again for local confluence. If this process eventually causes $\mathcal{R}$ to be locally confluent, and no unorderable critical pairs were found, the resulting rewriting system is complete (convergent), and has the same equational theory as the original.

The Huet's Knuth-Bendix completion procedure uses the above method for constructing a complete TRS $\mathcal{R}$ given a set of equations $\mathcal{E}$. Figure 3.1, contains the completion procedure as described in [HUE 81]. In this figure, **repeat** means "go to the first statement of the smallest enclosing loop." The procedure in Figure 3.1 makes use of the following functions: $Select(\mathcal{E}, s = t)$ means choose any equation $s = t$ from $\mathcal{E}$, $Normal(t, t', \mathcal{R})$ calculates a normal form t' of a term t, $Order(s, t)$ is equivalent to "**if** $s \succ t$ **then** $s \rightarrow t$ **else if** $t \succ s$ **then** $t \rightarrow s$ **else** fails," and finally $CriticalPairs(r, r')$ calculates the set of all critical pairs between the rules r, r'. The initial input to the procedure is a reduction ordering $\succ$, and a set of equations $\mathcal{E}$. At the beginning the rewriting system $\mathcal{R}$ is an empty set. The resulting rewriting system is represented by a set of triples. Each triple consists of integer label, a rewrite rule, and a flag, in that order, if the flag is "true", the rewrite rule is considered to be "marked" (when its critical pairs with all other rules are computed); if the flag is "false", the rule is "unmarked".

When looking for a decision procedure for an equational theory $=_{\mathcal{E}}$ using the Knuth-Bendix completion procedure in Figure 3.1, first one selects a reduction ordering $\succ$ and a set of equations, and then one executes the procedure. The procedure may halt with "failure" if the two sides of a rule are not orderable, or may fail to terminate because it may generate an infinite set of rules. Since any

```
R = ∅; n = 0
loop
    while E ≠ ∅ do

        Treat critical pair:
        Select(E, s = t)
        E = E − {s = t}
        Normal(s, s', R)
        Normal(t, t', R)
        if s' = t' then repeat endif

        Order equation:
        if not Order(s', t') then stop withfailure endif
        (λ → ρ) = Order(s', t')

        Normalize rewriting system:
        for each ⟨γ → μ, i, flag⟩ in R do
            Normal(γ, γ', {λ → ρ, -, -})
            if γ ≠ γ' then
                R = R − {⟨γ → μ, i, flag⟩} ; E = E ∪ {γ' = μ}
            else
                Normal(μ, μ', R ∪ {⟨λ → ρ, -, -⟩}
                if μ ≠ μ' then R = (R − {⟨γ → μ, i, flag⟩}) ∪ {⟨γ → μ', i, flag⟩} endif
            endif
        endfor

        n = n + 1
        R = R ∪ {⟨λ → ρ, n, false⟩}
    endwhile

    Find an unmarked rule:
    for each ⟨λ → ρ, i, flag⟩ in R do
        if flag = false then goto Computecriticalpair endif
    endfor
    stop withsuccess

    Compute critical pair:
    for each ⟨γ → μ, k, flag⟩ in R do
        if k ≤ i then E = E ∪ CriticalPairs(λ → ρ, γ → μ) endif
    endfor

    Mark the rule:
    R = R − {⟨λ → ρ, i, false⟩} ∪ {⟨λ → ρ, i, true⟩}
endloop
```

Figure 3.1 Huet's Knuth-Bendix Completion Procedure

practical implementation needs a means to stop, this may be provided by setting a limit on the number of iterations of the main loop.

In the above procedure, when $\succ$ is unable to order the two sides of a rewrite rule, it is either because $\succ$ is not general enough to show that $\mathcal{R}$ terminates, or because the rule is inherently non-terminating, (i.e. $+(x,y) \rightarrow +(y,x)$). This is one of the major drawbacks of the Knuth-Bendix completion procedure as presented here: it does not apply to theories containing such (useful) permutative equations. However there exist some extensions of this procedure which deals with this kind of equations, and they are presented in [HUE 80b] and [PET 81].

This procedure has been succefully used on a number of interesting axioms sets. One interesting, and often referenced, such example is the one of group theory, as defined in § 3.3. As indicated in [HUE 80a], and [HUL 80a], the completed rewriting system is shown in Figure 3.2. The completion of this system is shown in Appendix A using our conditional Knuth-Bendix procedure implementation described in Chapter 4. The first three rules in Figure 3.2 are the original axioms. Note that the equational theorem $i(i(x))$, which we proved by hand in § 3.3, was explicitly generated as rule 6 in Figure 3.2. For example rule (4) in Figure 3.2 is generated by superposing the left-hand side of rule (3) at occurrence 1 with the left-hand side of rule (2).

(1) $+(0,x) \rightarrow x.$
(2) $+(i(x),x) \rightarrow 0.$
(3) $+(+(x,y),z) \rightarrow +(x,+(y,z)).$
(4) $+(i(x),+(x,z)) \rightarrow z.$
(5) $i(0) \rightarrow 0.$
(6) $i(i(x)) \rightarrow x.$
(7) $+(x,0) \rightarrow x.$
(8) $+(x,i(x)) \rightarrow 0.$
(9) $+(x,+(i(x),z)) \rightarrow z.$
(10) $i(+(x,y)) \rightarrow +(i(x),i(y)).$

Figure 3.2 A Complete Rewriting System for Group Theory

To prove an equational theorem using a term rewriting system $\mathcal{R}$, one may reduce the left-hand side and the right-hand side of the equation to normal forms:

if the two normal forms are identical then the equation is an equational theorem of $=_{\mathcal{R}}$, otherwise the equation is not an equational theorem of $=_{\mathcal{R}}$. For example, to prove the equational theorem

$$(E3) \qquad i(+(i(x), i(y))) = +(y, i(+(i(x), 0)))$$

using the rewriting system in Figure 3.2, we proceed as follows: first reduce the left-hand side of the equation to obtain $+(y, x)$ by applying rules (10,6,and 6) in that order, then reduce the right-hand side to obtain $+(y, x)$ after applying respectively rules (10,5,1,and 6). Since the two resulting normal forms are identical, then equation $(E3)$ is an equational theorem of groups (this procedure is effective only when the rewriting system is terminating, see last section).

In addition to its use as a means of obtaining decision procedures for equational theories and for the construction of rewriting systems that can be used by abstract interpreters,the Knuth-Bendix procedure may be also used, to prove theorems by refutation [HSI 82]; to compute congruence closures of finite sets of ground equations [LAN 75b], and to verify and synthesize "rewrite programs" [DER 83a]. See [DER 83b] for a survey of these applications.

3.6 Conditional Term Rewriting Systems

3.6.1 General Results

A *conditional Term Rewriting System (conditional TRS)* is a finite set of conditional rewrite rules, with the following form:

$$\bigwedge_{i=1}^{n} p_i = q_i \Rightarrow \lambda \rightarrow \rho.$$

satisfying the following conditions:

a) $\mathcal{V}(p_i) \subseteq \mathcal{V}(\lambda)$.

b) $\mathcal{V}(q_i) \subseteq \mathcal{V}(\lambda)$.

c) $\mathcal{V}(\rho) \subseteq \mathcal{V}(\lambda)$.

The $(\bigwedge_{i=1}^{n} p_i = q_i)$ part is called the premises of the rule, and the $\lambda \rightarrow \rho$ part is called its conclusion. Note that when $n = 0$ the premises are empty, and we have an ordinary rewrite rule, so term rewriting systems are a subcase of conditional term rewriting system.

Given a conditional TRS $\mathcal{R}_c$ and two terms s and t, we write $s \rightarrow_{\mathcal{R}_c} t$ if and only if there exists a rule $\bigwedge_{i=1}^{n} p_i = q_i \Rightarrow \lambda \rightarrow \rho$ in $\mathcal{R}_c$ such that: (a) there exists a substitution σ and an occurence $u \in \mathcal{O}(s)$ such that $s/u = \sigma(\lambda)$, and $t = s[u \leftarrow \sigma(\rho)]$, (b) for all $i \in [1, \ldots, n]$ there exists γ_i such that $\sigma(p_i) \rightarrow^*_{\mathcal{R}_c} \gamma_i$, and $\sigma(q_i) \rightarrow^*_{\mathcal{R}_c} \gamma_i$. In other words, a conclusion $\lambda \rightarrow \rho$ of a conditional rule is used to rewrite a term s into t only if its premises are verified, which is recursively determined by rewriting.

Note that other interpretations of $\rightarrow_{\mathcal{R}_c}$ are possible. This is discussed in [BER 82],[KAP 84a]. Now that we have the definition of $\rightarrow_{\mathcal{R}_c}$, we can use the notation introduced in §3.5 without redefinitions ($\rightarrow_{\mathcal{R}_c}$, $\rightarrow^*_{\mathcal{R}_c}$, $\leftrightarrow^*_{\mathcal{R}_c}$, $=_{\mathcal{R}_c}$, $\downarrow$) in the conditional case.

The main problem regarding conditional TRS is that in order to apply a rule, one intuitively has to recursively evaluate the premises of the rule by rewriting, which may lead to infinite loops even for one step of rewriting. This is stated by the following theorem:

Theorem 3. *There exists a conditional TRS, which is confluent and neotherian, but such that the one step rewriting of a term is non-decidable and the normal form is non-computable.*

An example of such TRS appears in [KAP 84a]. In order to deal with the termination of the premises, we will use the the concept of *fair conditional TRS* which was first introduced in [KAP 84b]. Recall that the point is to avoid infinite calls to the evaluation procedure when the premises have to be evaluated. This may be achieved if the premises of each rule are "simpler", in some sense, than its conclusion. This can be done using a reduction ordering $\succ$, such that for every rule $\bigwedge_{i=1}^{n} p_i = q_i \Rightarrow \lambda \rightarrow \rho$, in $\mathcal{R}_c$ we have (a) $\lambda \succ \rho$, (b) $\lambda \succ p_i$ and $\lambda \succ q_i$. When these conditions are satisfied the conditional TRS $\mathcal{R}_c$ is noetherian (terminating) and $\mathcal{R}_c$ is also one-step terminating, which means that during the the normal form computation for a term t, there is no infinite sequence of calls to the evaluation procedure. Moreover when $\mathcal{R}_c$ is confluent, the reduction $(\rightarrow_{\mathcal{R}_c})$ is totally decidable and the normal form computation is correct relative to $\mathcal{R}_c$. These results have been proved in [KAP 84b]. Term rewriting systems that satisfy these conditions are called *fair conditional TRSs*. These requirements are sufficient to recover tractability for one-step rewriting and normal form function computation. We shall see in the following section that these requirements allow one to deal with several questions addressed in the field of term rewriting systems: the Knuth-Bendix criterion for confluence, completion procedure, ... These questions have well-known answers in the non-conditional case as presented in §3.4 and §3.5. From now on, we restrict our discussion to fair conditional TRSs. As an example of fair conditional TRS, see Figure 3.3.

(1) $succ(pred(x)) \rightarrow x.$
(2) $pred(succ(x)) \rightarrow x.$
(3) $leq(zero, zero) \rightarrow true.$
(4) $leq(zero, pred(zero)) \rightarrow false.$
(5) $leq(zero, x) = true \Rightarrow leq(zero, succ(x)) \rightarrow true.$
(6) $leq(zero, x) = false \Rightarrow leq(zero, pred(x)) \rightarrow false.$
(7) $leq(succ(x), y) \rightarrow leq(x, pred(y)).$
(8) $leq(pred(x), y) \rightarrow leq(x, succ(y)).$

Figure 3.3 A Fair Conditional TRS Example

This TRS is a fair conditional TRS. In [KAP 84a] it is shown that this system models the integers with "less or equal" predicate. It shown in Appendix A using the conditional Knuth-Bendix completion procedure described in Chapter 4, that this system is a confluent fair conditional TRS. Thus it is a complete conditional TRS.

3.6.2 A Naive Conditional Knuth-Bendix Completion Procedure

Let $\mathcal{R}_c$ be a fair conditional TRS. Suppose that the classical Knuth-Bendix completion procedure of § 3.5 is run on $\mathcal{R}_c$ considering only the conclusion part of rules in $\mathcal{R}_c$: then if the $\lambda_i \rightarrow \rho_i$ part of two rules: $\bigwedge_{i=1}^{m_1} p_{1,i} = q_{1,i} \Rightarrow \lambda_1 \rightarrow \rho_1$ and $\bigwedge_{i=1}^{m_2} p_{2,i} = q_{2,i} \Rightarrow \lambda_2 \rightarrow \rho_2$ generates a critical pair $\langle \lambda, \rho \rangle$ (oriented into $\lambda \rightarrow \rho$) via a substitution σ, the new rule $\sigma((\bigwedge_{i=1}^{m_1} p_{1,i} = q_{1,i}) \wedge (\bigwedge_{i=1}^{m_2} p_{2,i} = q_{2,i})) \Rightarrow \lambda \rightarrow \rho$ is added to $\mathcal{R}_c$.

Suppose that this procedure stops, yielding a system $\mathcal{R}_c'$. Then $\mathcal{R}_c'$ is confluent and $=_{\mathcal{R}_c} = =_{\mathcal{R}_c'}$. In that sense, this completion procedure is correct. Nevertheless, it would generate too large a system of rules, that would run forever too frequently, and produce irrelevant critical pairs. Kaplan [KAP 84b] has proposed another approach taking into account the treatment of premises. We consider it as an optimization of the procedure described hereabove. In this procedure, whenever critical pairs are found, we form the premises of the added equation by taking the union of the premises of the rules used to get the critical pairs. Also this procedure must discard rewrite rules with non-unifiable premises in the algebraic theory defined by $\mathcal{R}_c$, so we would like to possess some rules for evaluating premises. In fact we would like to solve the premises in the algebraic theory, which requires that rewriting is done via a unifier rather than a matcher. For this purpose Kaplan [KAP 84b] developed

a procedure to decide in some case this problem. This is the subject of the next section.

3.6.3 Unification in Conditional Theories

We suppose that a fair conditional TRS $\mathcal{R}_c$ is given, to which the $=_{\mathcal{R}_e}$ on T is associated. Given two terms s and t, we consider the following question: find all substitutions σ such that $\sigma(s) =_{\mathcal{R}_e} \sigma(t)$. (Such a σ is called a *unifier* of s and t in the *algebraic theory* defined by $\mathcal{R}_c$.)

This definition is extended to *clauses*: given a clause (C) $\wedge_{i=1}^{n} p_i = q_i$, C is $\mathcal{R}_c$-*unifiable* if and only if there exists a substitution σ such that

$$\sigma(p_1) =_{\mathcal{R}_e} \sigma(q_1) \text{ and } \ldots \sigma(p_n) =_{\mathcal{R}_e} \sigma(q_n).$$

For the case of equational (non-conditional) theories, a satisfying answer to the problem of finding a unifier relative to the TRS may be found in [FAY 77], and [HUL 80]. A semi-decision procedure, based on the notion of *narrowing*, is described, that gives a complete (and possibly infinite) set of such unifiers. Here we present the extension of this method to the case of fair conditional TRS as in [KAP 84b].

In the equational (non-conditional) case the central idea for the notion of narrowing is the following: given a rule $r : \lambda \rightarrow \rho$, we say that a term t is *narrowable* to t' at occurrence u using rule r via a substitution σ, and we write: $t \rightsquigarrow_{u,r,\sigma} t'$, if $\sigma(t/u) = \sigma(\lambda)$ and $t' = \sigma(t[u \leftarrow \rho])$ (σ being the *mgu* for t/u and λ). Note that narrowing is different from rewriting since in the case of rewriting the substitution σ is only the matcher of t/u by λ as defined in §3.2, while in the case of narrowing σ is the unifier.

Consider two terms s and t in T that are $\mathcal{R}$-unifiable (where $\mathcal{R}$ is a non-conditional TRS), i.e. there exists a substitution σ such that $\sigma(s) =_{\mathcal{R}} \sigma(t)$. Thus:

- either σ is the *mgu* of s and t, and the classical unification algorithms give the *mgu* of s and t.
- or one of $\sigma(s)$ or $\sigma(t)$ is $\mathcal{R}$-reducible, i.e. there exists a rule in $\mathcal{R}$ such that $\sigma(s)$ or $\sigma(t)$ is reducible using that rule (otherwise, it would be impossible to have $\sigma(s) =_{\mathcal{R}} \sigma(t)$). In that case, suppose that $\sigma(s)$ may be reduced by the rule $r : \lambda \rightarrow \rho$ (we assume that variables in λ and s are renamed to eliminate common variables). For a suitable occurrence u in s and a substitution σ', one has $\sigma(s/u) = \sigma'(\lambda)$. Thus s is narrowable using the rule r via substitution $\sigma \circ \sigma'$ at occurence u.

Consider now the conditional case. As previously, suppose that $\sigma(s)$ is reducible by a rule $r : \wedge_{i=1}^{n} p_i = q_i \Rightarrow \lambda \rightarrow \rho$ with $\sigma(s/u) = \sigma'(\lambda)$. The main difference with the equational case is that one has the following relation: $\wedge_{i=1}^{n} \sigma'(p_i) =_{\mathcal{R}_c} \sigma'(q_i)$ (*). (The relation (*) must be true because in the conditional case a rule can be used to reduce a term only if its premises are verified). As previously, s and λ are unifiable. Let μ be their *mgu*. There exists γ such that $\sigma' = \mu \circ \gamma$. The relation (*) may be interpreted as: the premises of the rule r are $\mathcal{R}_c$-unifiable (via γ). This leads to the following definition:

Definition 1. *[KAP 84b] Given two clauses C and C', the couple (t, C) is* **narrowable** *to (t', C') at occurrence u using the rule $r : \wedge_{i=1}^{n} p_i = q_i \Rightarrow \lambda \rightarrow \rho$, via a substitution σ, i.e:*

$$(t, C) \rightsquigarrow_{u,r,\sigma} (t', C'),$$

if

(1) t/u, and λ are unifiable (with mgu σ),

(2) $t' = \sigma(t[u \leftarrow \rho])$, and

(3) $C' = C \cup \wedge_{i=1}^{n} \sigma(p_i) = \sigma(q_i)$.

For example, consider the fair conditional TRS of Figure 3.3, and let $t = leq(zero, succ(succ(y)))$ and C be the empty clause. Then the couple (t, C) is narrowable to the couple (t', C') where $t' = true$, and $C' = (leq(zero, succ(y)) = true)$ using rule (5) of Figure 3.3 and via a substitution $\sigma = \{x \leftarrow succ(y)\}$.

Given a fair conditional TRS $\mathcal{R}_c$, Kaplan [KAP 84b] has developed a procedure based on the above definition to test if two terms are $\mathcal{R}_c$-unifiable. This procedure is shown in Figure 3.4. The procedure accepts two terms and calculates their $\mathcal{R}_c$-unifier if it exists. It uses the function $unify(s, t, \mu)$ to calculate the unifier μ of the two terms s and t. The $unify\_conj(C)$ function is used to check whether a clause C is unifiable or not, this consists in finding a unifier for which each two sides of an equation in the clause are unifiable relative to $\mathcal{R}_c$. The binary operator $\oplus$ is used only for technical reasons to simplify the procedure. It takes two terms and it forms a pair of them. As in Figure 3.1, **repeat** means "go to the first statement of the smallest enclosing loop."

It is shown in [KAP 84b] that this procedure is a semi-decision procedure for the computation of the $\mathcal{R}_c$-unifer of two terms. This procedure will be used by the Knuth-Bendix-like procedure for fair conditional TRS described in §3.6.4.

3.6.4 Confluence for Fair Conditional TRS Extending the Knuth-Bendix criterion

In §3.6.1, the question of global and one-step termination is dealt with using reduction ordering; fair conditional TRSs are shown to have these two properties. Now we will consider the problem of confluence of such systems. First, we state the following result that holds for neotherian conditional TRS in their full generality (i.e. without the hypothesis of one-step termination).

```
proc unify_{R_c}(s,t)
    C = {∅}, σ = ε
    T = ⊕(s,t)
    loop
        if unify(s,t,μ) then
            σ := σ ∘ μ stop WITH SUCCESS
        endif
        loop
          if (∃(label, ∧_{i=1}^n p_i = q_i ⇒ λ → ρ, false)) and unify(T/u, λ, σ')) then
                C := σ'(C ∪ ∧_{i=1}^n p_i = q_i)
                    if unify_conj(C) then
                      goto update
                    else repeat
                    endif
            else stop WITH FAILURE
        endloop
        update:
            T := σ'(T[u ← ρ])
            s := T/1, t := T/2
            σ = σ ∘ σ'
            repeat
    endloop
endproc
```

Figure 3.4 $\mathcal{R}_c$-unification Procedure

Theorem 4. *The problem of confluence for neotherian conditional TRS is undecidable.*

The proof of this theorem is given in [KAP 84a]. This result justifies the consideration of fair systems, for which a Knuth-Bendix-like criterion was provided. This criterion will be described later in this section.

3.6.4.1 The Conditional Knuth-Bendix Criterion

First, we need the following definitions. Given a fair conditional TRS $\mathcal{R}_c$, let $\bigwedge_{i=1}^{m_1} p_{1,i} = q_{1,i} \Rightarrow \lambda_1 \rightarrow \rho_1$ and $\bigwedge_{i=1}^{m_2} p_{2,i} = q_{2,i} \Rightarrow \lambda_2 \rightarrow \rho_2$, be two rules in $\mathcal{R}_c$ such that λ_1 and λ_2 overlap at occurence u in λ_1, and let σ be the *mgu* of λ_1/u and λ_2 (we assume that variables have been renamed to eliminate sharing between rules).

a) The formula:

$$\sigma((\bigwedge_{i=1}^{m_1} p_{1,i} = q_{1,i}) \wedge (\bigwedge_{i=1}^{m_2} p_{2,i} = q_{2,i})) \Rightarrow \langle \lambda_1[u \leftarrow \sigma(\rho_2)], \sigma(\rho_1) \rangle$$

is called a *contextual critical pair* (CCP for short) in the *contextual critical context:*

$$\sigma((\bigwedge_{i=1}^{m_1} p_{1,i} = q_{1,i}) \wedge (\bigwedge_{i=1}^{m_2} p_{2,i} = q_{2,i})).$$

b) A critical context $(\wedge_{i=1}^{m} p_i = q_i)$ is called *feasible* if and only if it is $\mathcal{R}_c$-unifiable, which means that there exists a substitution τ such that $\wedge_{i=1}^{m} \tau(p_i) = \tau(q_i)$. This substitution can be found by the procedure of Figure 3.4.

c) A CCP is said to be *feasible* if and only if its associated critical context is feasible.

Now, we will see by means of an example how this criterion on the CCP's may be used when testing for confluence. Consider the following system, $\mathcal{R}_c$:

$$1: \; even(0) \rightarrow True$$

$$2: \; even(succ(0)) \rightarrow False$$

$$3: \; even(succ(succ(x))) \rightarrow even(x)$$

$$4: \; even(x) = True \Rightarrow odd(x) \rightarrow False$$

$$5: \; even(x) = False \Rightarrow odd(x) \rightarrow True$$

Rule 1 through rule 3 provide a (non-conditional) definition of the even predicate on the integers. Rule 4 and 5 form a (conditional) definition of "odd" in term of the predicate "even". Considering the confluence of $\mathcal{R}_c$ there is just one critical pair (in the previous sense), namely the formula:

$$even(x) = True \wedge even(x) = False \Rightarrow \langle True, False \rangle.$$

The previous Knuth-Bendix completion procedure would thus fail in orienting this critical pair and stop. However if we consider the feasibility of the critical context associated with the critical pair: either this critical context may be narrowed via rule (1), yielding $(true = true \wedge true = false)$ which is not unifiable, or may be narrowed via rule (2), yielding $(false = true \wedge false = false)$ which is also not unifiable. Note that this is can also be narrowed using rule (3), yielding $(even(x) = true \wedge even(x) = false)$ which in turn is narrowable via rule (1) or (2), yeilding as above a non-unifiable clause. Recall that this may be decided by the procedure given in Figure 3.4. Thus this critical context is non-feasible. Then the above critical pair is a non-feasible CCP. It seems logical to consider only feasible CCP when testing confluence. Based on this we present results formulated by Kaplan [KAP 84b] stating connection between criteria on the CCPs, and the notions of confluence of fair conditional TRS.

Theorem 5. *(Knuth-Bendix theorem for fair conditional TRS) Given a fair conditional TRS $\mathcal{R}_c$, consider the two groups of properties:*

(i) $\mathcal{R}_c$ is locally confluent, and thus confluent, and,

(a) For every feasible CCP $(C) \Rightarrow \langle s, t \rangle$, we have $s \downarrow = t \downarrow$.

(b) For every feasible CCP $(C) \Rightarrow \langle s, t \rangle$, we have for all substitutions σ with $\models_{\mathcal{R}_c} \sigma((C)), \sigma(s) \downarrow = \sigma(t) \downarrow$.

then:

$$(a) \quad \textit{implies} \quad (b) \quad \textit{if and only if} \quad (i).$$

Moreover:

$$(i) \quad \textit{if and only if} \quad (s =_{\mathcal{R}_c} t \quad \textit{if and only if } s \downarrow = t \downarrow).$$

The above theorem is a simplified version of the one given in [KAP 84b] which contains other properties that are of no interest to us. The $\models_{\mathcal{R}_c}$ means that the clause C is $\mathcal{R}_c$-unifiable. This may be decided by the procedure of Figure 3.3.

Among the results of Theorem 5, it is the equivalence ((i) if and only if (b)) that constitutes the extension of Knuth-Bendix theorem to the conditional case. But a Knuth-Bendix-like procedure based on criterion (b) would make a too intensive use of $\mathcal{R}_c$-unification procedure of Figure 3.4. Its use would thus be unrealistic in many practical cases. For this reason, we will focus on a completion procedure based on criterion (a). Both procedures are described in [KAP 84b]. Figure 3.5 contains the completion procedure based on criterion (a). Compared with the naive completion procedure of §3.6.2, it is a very neat optimization, that remains realistic. Compared with the one based on criterion (b) it is less powerful, but much more realistic, and this justifies our choice.

The procedure in Figure 3.5 takes a set of conditional equations $\mathcal{E}_c$ as input, and produces, whenever it stops, a fair conditional TRS $\mathcal{R}_c$, which is confluent, thus complete, such that $=_{\mathcal{E}_c}=\leftrightarrow^*_{\mathcal{R}_c}$. The proof of the correctness of this procedure is given in [KAP 84b]. The author of this procedure mentions that to implement it, one has to use more efficient forms of it. This is presented in the next chapter where some improvements implemented on this procedure are described with the resulting procedure design.

3.7 Construction of Fair Conditional TRS

In this section, we present the Recursive Path Ordering (RPO) method for proving termination of term rewriting systems. This method has been implemented as a sub-module of the validator, and it is used by the conditional Knuth-Bendix implementation as described in Chapter 4 (see §4.3).

$\mathcal{R}_e = \emptyset$;
loop

if $\mathcal{E}_e = \emptyset$ **then stop** *withsuccess* **endif**

Treat contextual critical pairs:

Select$(\mathcal{E}_e, \wedge_{i=1}^{m} p_i = q_i \Rightarrow s = t)$
$\mathcal{E}_e = \mathcal{E}_e - \{\wedge_{i=1}^{m} p_i = q_i \Rightarrow s = t\}$
Normal$(\wedge_{i=1}^{m} p_i = q_i \Rightarrow s = t, \wedge_{i=1}^{m} p_i' = q_i' \Rightarrow s' = t', \mathcal{R}_e)$

Order formula:

if not *Order*$(\wedge_{i=1}^{m} p_i' = q_i' \Rightarrow s' = t')$ **then stop** *withfailure* **endif**

$(\wedge_{i=1}^{m} u_i = v_i \Rightarrow \lambda \rightarrow \rho) = Order(\wedge_{i=1}^{m} p_i' = q_i' \Rightarrow s' = t')$

Normalize rewriting system:

for each $(\wedge_{i=1}^{m} p_i = q_i \Rightarrow \gamma \rightarrow \mu)$ **in** $\mathcal{R}_e$ **do**
  Normal$(\wedge_{i=1}^{m} p_i = q_i \Rightarrow \gamma \rightarrow \mu, \wedge_{i=1}^{m} p_i' = q_i' \Rightarrow \gamma' \rightarrow \mu', \{\wedge_{i=1}^{m} u_i = v_i \Rightarrow \lambda \rightarrow \rho\})$
  if $\gamma \not\equiv \gamma'$ **or** $\mu \not\equiv \mu'$ **then**
    $\mathcal{R}_e = \mathcal{R}_e - \{\wedge_{i=1}^{m} p_i = q_i \Rightarrow \gamma \rightarrow \mu\}$
    $\mathcal{E}_e = \mathcal{E}_e \cup \{\wedge_{i=1}^{m} p_i' = q_i' \Rightarrow \gamma' \rightarrow \mu'\}$
  endif
endfor

$\mathcal{R}_e = \mathcal{R}_e \cup \{\wedge_{i=1}^{m} u_i = v_i \Rightarrow \lambda \rightarrow \rho\}$

Compute contextual critical pairs:

for each $(\wedge_{i=1}^{m} p_i = q_i \Rightarrow \gamma_1 \rightarrow \mu_1)$ **in** $\mathcal{R}_e$ **do**
  for each $(\wedge_{i=1}^{m} u_i = v_i \Rightarrow \lambda_1 \rightarrow \rho_1)$ **in** $\mathcal{R}_e$ **do**
    $CCP = \{CriticalPairs(\wedge_{i=1}^{m} p_i = q_i \Rightarrow \gamma_1 \rightarrow \mu_1, \wedge_{i=1}^{m} u_i = v_i \Rightarrow \lambda_1 \rightarrow \rho_1)\}$
  endfor
endfor

Check for feasible critical pairs:

for each $|(C) \Rightarrow (s,t)|$ **in** CCP **do**
  if *unify_conj*(C) **then**
    $\mathcal{E}_e = \mathcal{E}_e \cup \{(C) \Rightarrow s = t\}$
  endif
endfor

endloop

Figure 3.5 Kaplan's Conditional Knuth-Bendix Completion Procedure

Termination is undecidable. Nevertheless, we are often interested in whether

a conditional rewriting system $\mathcal{R}_c$ is fair i.e. noetherian and one-step terminating, because (see §3.6)

- Fairness allows one to decide whether $\mathcal{R}_c$ is confluent.
- If $\mathcal{R}_c$ is confluent, fairness allows one to decide $=_{\mathcal{R}_c}$.
- If $\mathcal{R}_c$ is not confluent, fairness allows one to use the Knuth-Bendix-like completion procedure to help achieve confluence.

The Knuth-Bendix-like procedure, as part of the completion process, constructs a terminating rewriting system from a set of conditional equations with the use of a reduction ordering $\succ$. The construction process consists of showing that every conditional equation can be ordered, in one direction or the other, into a conditional rewrite rule $\wedge_{i=1}^{n} q_i = p_i \Rightarrow \lambda \rightarrow \rho$, such that for all $i \in [1 \ldots n]$ $\lambda \succ p_i$ and $\lambda \succ q_i$, and $\lambda \succ \rho$. These ordered conditional rewrite rules comprise $\mathcal{R}_c$, and $\succ$ proves that $\mathcal{R}_c$ terminates.

In what follows: §3.7.1 presents the basic definitions and theory behind the use of orderings in constructing terminating conditional rewrite rules. §3.7.2 describes the recursive path ordering method. Note that here we do not treat non-conditional term rewriting systems separately because they are a sub-case of conditional term rewriting systems. Thus all the results that hold for conditional TRSs hold also for TRSs.

3.7.1 Orderings Definitions and Properties

3.7.1.1 Relations and Orderings

This section is concerned with various binary relation. A *binary relation* φ is a set of ordered pairs of elements belonging to a *base set* S. The notation $s\varphi t$ means $\langle s, t \rangle \in \varphi$.

A *relation pair* is inductively defined to be a pair $\langle \varphi_1, \varphi_2 \rangle$, where φ_1 and φ_2 are either relations or relation pairs. The *base set* of $\langle \varphi_1, \varphi_2 \rangle$ is the union of the base sets of φ_1 and φ_2.

A *quasi ordering* $\succeq$ is a transitive, reflexive binary relation. The notation $s \simeq t$ means ($s \succeq t$ and $t \succeq s$), and $s \not\simeq t$ means $\langle s,t \rangle \notin \succeq$. We say that s and t are *comparable* under $\succeq$ if and only if $s \succeq t$ or $t \succeq s$.

A *partial ordering* $\succ$ is a transitive, irreflexive binary relation. The notation $s \not\succ t$ means $\langle s,t \rangle \notin \succ$. We can obtain a partial ordering $\succ$ from a quasi ordering $\succeq$, by defining $s \succ t$ if and only if ($s \succeq t$ and $t \not\simeq s$). We say that a partial ordering $\succ$ is *well-founded* if and only if it admits no infinite descending sequences $s_1 \succ s_2 \succ s_3 \succ \ldots$ of elements in its base set. An *ordering* is a quasi or partial ordering.

3.7.1.2 Simplification orderings

Dershowitz [DER 82] introduced a general class of partial orderings on terms, known as *simplification orderings*, and showed that simplification orderings can be straightforwardly used to prove the termination of rewriting systems.

Definition 2. *A partial ordering $\succ$ on terms is a simplification ordering if it possesses the following two properties:*

Computability: $s \succ t \Rightarrow f(\ldots s \ldots) \succ f(\ldots t \ldots)$

Subterm: $f(\ldots t \ldots) \succ t$

for any terms s, t, $f(\ldots s \ldots)$, and $f(\ldots t \ldots)$.

Kaplan [KAP 84b] extended Dershowitz idea to the framework of conditional rewriting systems.

Theorem 6. *[KAP 84b] A conditional TRS $\mathcal{R}_c$ is fair if there exists a simplification ordering $\succ$, such that $\sigma(\lambda) \succ \sigma(\rho)$ and $\forall i \in [1 \ldots n]$ $\sigma(\lambda) \succ \sigma(p_i)$ and $\sigma(\lambda) \succ \sigma(q_i)$ for all substitutions σ and all rules $\wedge_{i=1}^{n} p_i = q_i \Rightarrow \lambda \rightarrow \rho$ in $\mathcal{R}_c$.*

The above theorem is taken from [KAP 84b], which may be considered as extending Dershowitz's [DER 82].

The simplification orderings that we consider here are *stable*; i.e. $s \succ t$ implies $\sigma(s) \succ \sigma(t)$, for all terms s and t, and all substitutions σ. Consequently, we may use a variant of the above theorem that is slightly less general.

Theorem 7. *A conditional TRS $\mathcal{R}_c$ is fair if there exists a stable simplification ordering $\succ$, such that $\lambda \succ \rho$ and $\forall i \in [1 \ldots n]$ $\lambda \succ p_i$ and $\lambda \succ q_i$ for all rules $\wedge_{i=1}^{n} p_i = q_i \Rightarrow \lambda \rightarrow \rho$ in $\mathcal{R}_c$.*

In §3.6.1, we indicated that the Knuth-Bendix-like procedure uses a reduction ordering to prove fairness. Theorem 7 indicates that stable simplification orderings can be used instead. Stable simplification orderings are different from reduction orderings, because simplification orderings are not necessarily well-founded, and reduction orderings do not necessarily have the subterm property. However, Dershowitz [DER 82] showed that when the base sets of the orderings are restricted to terms over a finite set of operators, such as terms that comprise a (finite) conditional TRS which we are dealing with, the simplification ordering is the same as the reduction ordering. In applications other than termination proofs, when a well-founded ordering is needed for terms over an infinite set of operators, one must separately show the well-foundedness of the simplification ordering. See [DER 83c] for techniques in constructing well-founded orderings, and for an overview of most known classes of simplification orderings, including the one discussed here.

3.7.2 The Recursive Path Ordering

Before giving the definition of the Recursive Path Ordering (RPO), we first need some subsidiary definitions.

A *precedence* $\pi = \langle \geq^{\cdot}, \neq^{\cdot} \rangle$ is a relation pair, where $\geq^{\cdot}$ and $\neq^{\cdot}$ are binary relations on operators.

We say that f and g are *comparable* under $\langle \geq^{\cdot}, \neq^{\cdot} \rangle$ if and only if they are comparable under $\geq^{\cdot}$. We will use $f >^{\cdot} g$ as a shorthand for ($f \geq^{\cdot} g$ and $f \neq^{\cdot} g$), and $f \doteq g$ as a shorthand for ($f \geq^{\cdot} g$ and $g \geq^{\cdot} f$). Note that $>^{\cdot}$ is a partial ordering. We say that $\langle \geq^{\cdot}, \neq^{\cdot} \rangle$ is *total* if and only if, for all operators f and g in the base set, either $f >^{\cdot} g$, $g >^{\cdot} f$, or $f \doteq g$. The precedence $\langle \geq^{\cdot}, \neq^{\cdot} \rangle$ is total over T if it is total when its base set is restricted to T. We will usually use just π to denote a precedence, rather than $\langle \geq^{\cdot}, \neq^{\cdot} \rangle$.

Intuitively, a *multiset* m on a quasi ordering $\succeq$ is an unordered collection of elements, where m may contain multiple elements that are equivalent under $\simeq$. More formally, m is a mapping from the base set S of $\succeq$ onto the non-negative integers, that associates, with each member of S, the number of elements to which it is $\simeq$ in the multiset. We use $\{s_1, \ldots, s_n\}$ to denote the multiset containing the (possibly duplicated) elements $s_1, \ldots, s_n$. $\mathcal{M}(S)$ denotes the set of all finite multisets on S.

Definition 3. *[HUE 80a] Given a quasi ordering $\succeq$, whose base set is S, and elements m and m' of $\mathcal{M}(S)$, we obtain a relation $\succeq^{\mu}$ on $\mathcal{M}(S)$ defined as follows: $m \succeq^{\mu} m'$ if and only if $(\forall x)([m'(x) > m(x)]) \Rightarrow (\exists y)([y \succ x] \wedge [m(y) > m'(y)])$. The instantiations of $\succeq^{\mu}$ are quasi orderings, and are called the multiset orderings.*

See [JOU 82b] for properties of this ordering, a comparison of this ordering with other multiset orderings, and an efficient implementation. For example, if $>$ is the

'greater than' ordering on the natural numbers, then $\{3,3,4,0\} >^{\mu} \{3,2,2,1,1,1,4\}$ in the multiset ordering, since an occurrence of 3 has been replaced by five smaller numbers and in addition an occurrence of 0 has been removed (i.e. replaced by zero occurrences).

Now we are ready to give the definiton of the *Recursive Path Ordering* (*RPO*).

Definition 4. *Let π be a precedence on a set of operators F. The recursive path ordering $\succ_{\text{rpo}}$ on the set of terms over F is induced by the quasi ordering $\succeq_{\text{rpo}}$, where*

$$s = f(s_1, \ldots, s_m) \succeq_{\text{rpo}} g(t_1, \ldots, t_n) = t$$

is defined inductively as the union of the following three cases:

(1) $$(f \doteq g) \;\&\; (\{s_1, \ldots, s_m\} \succ^{\mu}_{\text{rpo}} \{t_1, \ldots, t_n\})$$

(2) $$(f \succ^{\cdot} g) \;\&\; (\forall t_i)(s \succ_{\text{rpo}} t_i)$$

(3) $$(\exists s_i)(s_i \succeq_{\text{rpo}} t)$$

where $\succeq^{\mu}_{\text{rpo}}$ is the extension of $\succeq_{\text{rpo}}$ to multisets.

Theorem 8. *The recursive path ordering $\succ_{\text{rpo}}$ is a simplification ordering.*

The proof of this theorem is given in [DER 82].

Since the recursive path ordering is a simplification ordering, it can be lifted to a stable ordering on terms with variables by treating variables as constants, where: i) $x \doteq x$ for all variables x, and ii) $\langle x, y\rangle \notin \geq^{\cdot}$ and $\langle x, y\rangle \notin \neq^{\cdot}$ for all distinct symbols x and y, where x and/or y is a variable. Therefore, it can be used in conjunction with Theorem 6 to prove the termination of conditional TRS. Furthermore the RPO

is well-founded if and only if the $>^{\cdot}$ on the set of operators F is well-founded. The partial ordering $>^{\cdot}$ will always be well-founded if its base set of operators is finite, but this coincides with our situation, hence the $\succ^{\text{rpo}}$, can be used by the conditional Knuth-Bendix to prove termination.

Now we show an example of use of the recursive path ordering to prove termination. Consider the following system for computing the disjunctive form of a logical formula:

$$not(not(x)) \rightarrow x.$$
$$not(or(x,y)) \rightarrow and(not(x), not(y)).$$
$$not(and(x,y)) \rightarrow or(not(x), not(y)).$$
$$and(x, or(y,z)) \rightarrow or(and(x,y), and(x,z)).$$
$$and(or(y,z), x) \rightarrow or(and(y,x), and(z,x)).$$

We wish to prove that this system terminates for all inputs. For this we use the recursive path ordering on terms with operators *not*, *and*, and *or* and a precedence π on these operator such that $not >^{\cdot} and <^{\cdot} or$. Since this is a stable simplification ordering on terms, by Theorem 7, we need only to prove that

$$not(not(x)) >_{\text{rpo}} x.$$
$$not(or(x,y)) >_{\text{rpo}} and(not(x), not(y)).$$
$$not(and(x,y)) >_{\text{rpo}} or(not(x), not(y)).$$
$$and(x, or(y,z)) >_{\text{rpo}} or(and(x,y), and(x,z)).$$
$$and(or(y,z), x) >_{\text{rpo}} or(and(y,x), and(z,x)).$$

for any terms x, y, and z.

The first inequality follows from the subterm condition of simplification orderings. By the definition of the recursive path ordering, to show that $not(or(x,y)) >_{\text{rpo}} and(not(x), not(y))$ when $not >^{\cdot} and$, we must show that $not(or(x,y)) >_{\text{rpo}} not(x)$

and $not(or(x,y)) >_{\text{rpo}} not(y)$. Now, since the outermost operators (root operators) of $not(or(x,y))$, $not(x)$, and $not(y)$ are the same, we must show that $or(x,y) >_{\text{rpo}} x$ and $or(x,y) >_{\text{rpo}} y$. But this is true by the subterm condition. Thus the second inequality also holds. By an analogous argument, the third inequality also holds.

For the fourth inequality, we must show $and(x,\ or(y,z)) >_{\text{rpo}} or(and(x,y),\ and(x,z))$. Since $and <\cdot\ or$, we must show $and(x,\ or(y,z)) >_{\text{rpo}} and(x,y)$ and $and(x,\ or(y,z)) >_{\text{rpo}} and(x,z)$. By the definition of the recursive path ordering for the case when two terms have the same outermost operator, we must show that $\{x, or(y,z)\} >^{\mu}_{\text{rpo}} \{x,y\}$ and $\{x, or(y,z)\} >^{\mu}_{\text{rpo}} \{x,z\}$. These two inequalities between multisets hold, since the element $or(y,z)$ is greater than both y and z with which it is replaced. Thus the fourth inequality holds. Similarly the fifth inequality may be shown to hold. Therefore, by Theorem 7, this system terminates for all inputs.

CHAPTER 4

SVELDA Design and Implementation

4.1 General Structure

In this Chapter, we describe the design and implementation of SVELDA (System for Validating and Executing LOTOS Data Abstractions). SVELDA was designed and implemented as a part of the LOTOS interpreter. It is composed of three main modules which in turn are composed of other sub-modules. Throughout this chapter module names will be written in **boldface.**

The general structure of SVELDA is shown in Figure 4.1. The first module is a **translator** from ACT ONE texts into a low-level specification acceptable by the **validator** and the **ADT interpreter** that constitute the two remaining main module. As shown in Figure 4.1 the **validator** and the **ADT interpreter** intersect. This intersection constitutes the basic sub-modules used by both the **validator** and the **ADT interpreter.** These sub-modules are described separately in this chapter, and references to those sub-modules are made where neccessary.

The **translator** was written in the C language. The **validator** and the **ADT interpreter** were instead written in PROLOG. PROLOG provides mechanisms for unification, backtracking, description of objects and relationships, and procedure abstraction. A PROLOG program consists of two parts: i) Facts that constitute the knowledge base ii) Rules that manipulate the facts in order to derive other facts. Throughout this chapter, by *packet* we mean a set of facts of the knowledge base with the same predicate name. The predicate name is used to identify the associated packet. In PROLOG facts may be added at the top or at the bottom of the packet. This is of great importance to us, because when we explore a packet in PROLOG

we always start at the top. The reason for choosing C to write the **translator** is that the latter was designed and implemented as a part of the "compiler" part of the LOTOS interpreter which was written in C. Various reasons motivated the use of PROLOG [CME 84] to write the **validator** and the **ADT interpreter**. Besides the obvious one, that it is a well-known language, widely used in research environments. The most important reason was the fact that the operational semantics of the LOTOS data part is defined in terms of rewrite rules, which can easily be programmed in PROLOG. Another good reason is that the **validator** and the **ADT interpreter** were designed and implemented to be used as a subordinate of the CCS* interpreter, which was also written in PROLOG. An example of usage of the **ADT interpreter** by the CCS* interpreter is that it evaluates ADT expressions when requested i.e. the CSS* interpreter invokes the **ADT interpreter** with an expression to be evaluated and the latter returns the normal form of that expression as a result.

The remaining of this chapter is organized as follows: §4.2 describes the **translator** and discusses the design and implementation choices that are made. §4.3 describes the basic modules that are common to the **validator** and the **ADT interpreter**. §4.4 describes the **validator** which mainly consists of an improved version of Kaplan's Knuth-Bendix-like procedure. The improvements implemented on this procedure are presented and discussed together with the organization of the resulting procedure, and other remaining modules of the **validator**. §4.5 describes the **ADT interpreter**.

4.2 Translator

A shown in Figure 1.1, the **translator** is a module of SVELDA that accepts the decorated LOTOS tree and the renamed symbol table as input and outputs a low-level data type specification in clausal form acceptable by PROLOG. The decorated

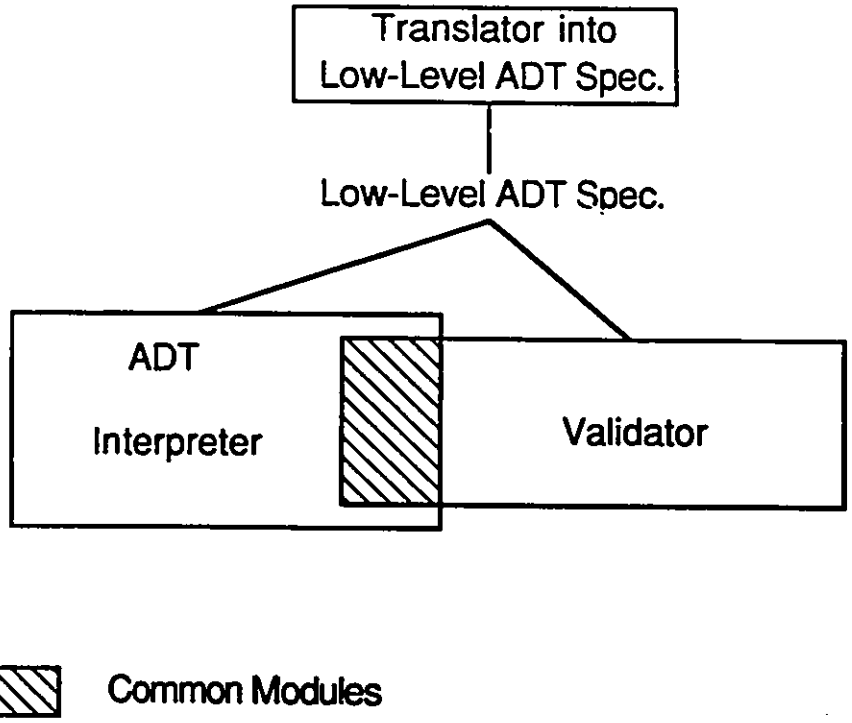

Figure 4.1 Structure of SVELDA Implementation

LOTOS tree contains among other things an internal representation of the data type definitions used in a LOTOS specification. The task of the **translator** is to produce a low-level data specification using the information stored in the tree and in the symbol table. The resulting data type definition is the union of all translated data type definitions used in a LOTOS specification.

The remaining of this section is organized as follows: §4.2.1 describes the tree representation of the data type definitions. §4.2.2 describes the translation process in detail, and gives an example of translation. §4.2.3 discusses the form of the data type specification after translation.

4.2.1 Tree Representation of Data Definitions

The data type definitions of a LOTOS specification are internally represented in the form of an ordered linked list. This ordering is done according to the dependency

of data type definitions in the specification. A node corresponding to a particular data type definition must appear before all the nodes of other data types that use it in their definitions. Usage of a type by others is to be understood as described in Chapter 2 (see §2.3).

One can see that the dependency of types in specifications is not necessarily total order, so we cannot obtain directly one ordered linked list that contains all the nodes of all the data type definitions used in a specification. To do this we have to extend the partial order to be a total order. This can be done easily because cycles in data type definitions are not allowed. In other words, one cannot have two data type definitions depending on each other i.e. a data type *t1* which uses a data type *t2* in its definition and vice versa. Furthermore, this ordering is strict since no data type definition may depend on itself.

The skeleton of this ordered linked list is produced by the syntax and static semantics analyser as shown in Figure 1.1. Figure 4.2 shows the structure of this list. The information stored in this list has the following properties:

- it is free of syntactic or static semantics errors.
- it is structured : this eliminates the need of sequentially scanning the input to get a specific information.

Each node in the ordered list corresponds to a data type definition used in a LOTOS specification, including those of the standard library. Each node has four pointers that point to a specific information item in the symbol table or the LOTOS decorated tree.

- *sy* : points to the symbol table and is used to identify the data type associated with the node.

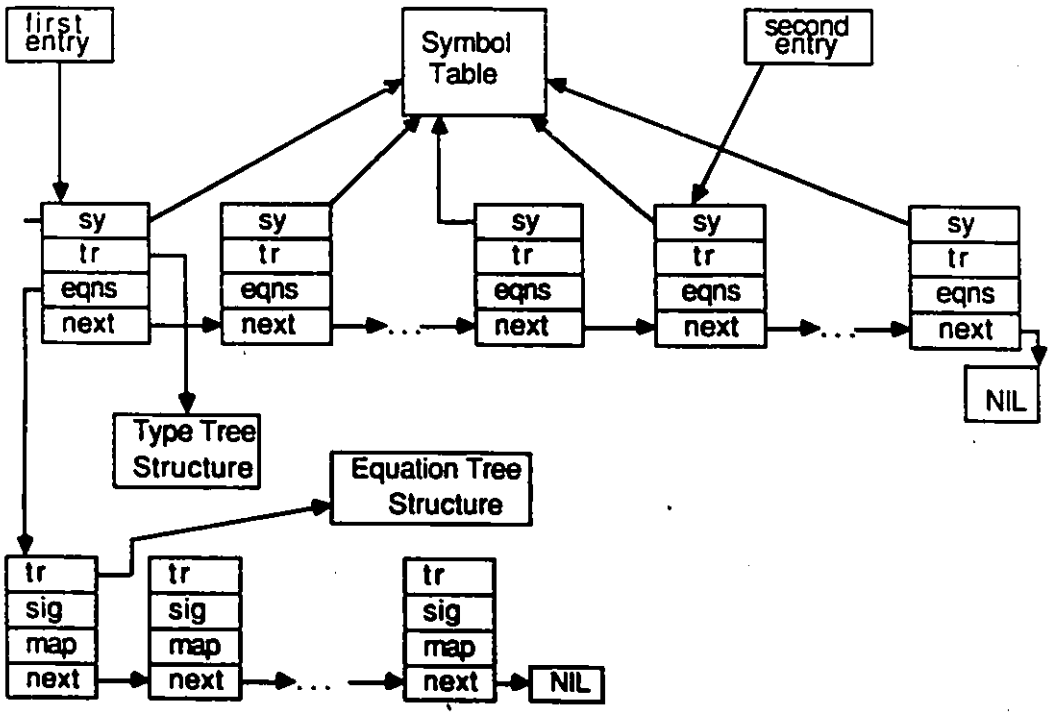

Figure 4.2 The Structure of the Ordered List

- ***tr*** **: points to the tree representation of the data type in the LOTOS decorated tree.**

- ***eqns*** **: points to the list of equations of the data type after translation.**

- *next* : points to the next node in the list.

Before translation, the *eqns* pointer points to an empty list for all the nodes in the ordered list. These lists of equations are calculated during the translation process which is described in the next section.

Finally, the ordered list has two entries : one corresponding to all the types defined by the specifier plus the types in the standard library. The other entry corresponds only to the types defined by the specifier. The reason for having two entries in the ordered list is that not all the data types in the standard library are to be included in the resulting data type specification, but only the ones that are used by some user-defined data types. So after calculating the list of equations for each user-defined type, the information corresponding to the types in the standard library is no longer needed. This is so because the list of equations calculated for each user-defined type will contain all equations of the data types used in its definition including the ones of the standard library. Hence, after the computations of all lists of equations the ordered list is explored starting at its second entry. This increases the efficiency of the translation process.

After this brief description of the internal structure of the data type definitions, we are now ready to describe the translation process.

4.2.2 The Translation Process

Before describing the process of translation, we first give an example of a LOTOS specification. Parts of this specification are used throughout this section each time a detailed explanation is needed to show some concepts used during translation. Consider the LOTOS specification skeleton of Figure 4.3:

In this specification, we assume that the standard library contains only three data types namely: $data, bool, queue(data)$. This is specified by the lib ... endlib

```
specif Example
    lib data, bool, queue(data) endlib
    type nat_num_queue is
        queue(data) actualizedby nat_num using
        sortnames nat for data
        opnames zero for d0
endtype

    type nat_num is
        sorts nat
        opns zero : -> nat
             succ : nat -> nat
    endtype

    process proc1[...] :=
        ...
        i; proc1[...]
        ...
    where
      type enriched_nat_num is nat_num
        opns plus : nat, nat -> nat
        eqns forall x, y : nat
        ofsort nat
            plus(x, zero) = x ;
            plus(x, succ(y)) = succ(plus(x, y)) ;
      endtype

      type connection(objects) is
        queue(data) renamedby
        sortnames channel for queue
                  objects for data
        opnames send for add
                receive for first
      endtype
    endproc
endspec
```

Figure 4.3. A LOTOS Specification Skeleton.

block. Two global data types are defined: *nat_num* and *nat_num_queue*. The specification contains also a process *proc1* that has a local data type definition composed of *enriched_nat*, and *connection(objects)*.

The ordered list corresponding to this specification before the translation contains a node for each data type definition including the ones of the standard library. The nodes in the ordered list may appear in this order: *data*, *bool*, *queue(data)*,

nat_num, *nat_num_queue*, *enriched_nat_num*, *connection*(*objects*). Note that even thought *nat_num* appears after *nat_num_queue* in the specification text, the node of the former appears before the node of the latter in the ordered list. This is due to the fact that *nat_num_queue* uses *nat_num* in its definition. Also note that for example *enriched_nat* and *connection*(*objects*) can exchange positions, since neither depends on the other. At this stage the lists of equations in all nodes are empty.

The translation process involves three steps that are done in sequence. The first step consists of travelling the whole ordered list, calculating the list of equations in each node in such a manner that duplication of equations is eliminated within each node (§4.2.2.1). In the second step a new node is created such that the list of equations equals the union of all the lists of equations corresponding to all the user-defined types (§4.2.2.2). This second step was done to eliminate duplicated equations that are not in the same list of equations, because the first step only eliminates duplicated equations belonging to the same list (note that equations that appear different in the specification may end up being the same because of renaming and/or actualization). Finally the third step is the output of the contents of the new node in a specific format (§4.2.2.3). Each step is described below in more detail.

4.2.2.1 Building the List of Equations

Each equation is represented by three elements: its signature, a mapping that maps the element in the signature to some others, and the tree structure of the equation itself as introduced in the input.

The signature of an equation is the set of operators used to build the equation and their sort. For any equation *eq* we will denote its signature by $\Sigma(eq)$. For

example the signature of the equation $plus(x, zero) = x$ in the list of equations of the *enriched_nat* is $\Sigma(plus(x, zero) = x) = \{plus, zero, nat\}$.

The mapping is the one corresponding to the replacement that is specified during a renaming of a data type definition or an actualization. See Chapter 2 for an introduction of these concepts. This mapping is empty if the equation belongs to neither a renamed type nor an actualized type. We denote this mapping by φ, and is represented by a set of pairs $\{(y, x), \ldots\}$. A pair (y, x) in this set indicates that the mapping function maps y to x. For example the mapping for the above equation in the *enriched_nat* is $\varphi = \{\emptyset\}$.

The computation of the list of equations for each node in the ordered list is done depending on the kind of the data type definition associated with that node. A data type can have one of three kinds: "normal" when the data type is defined using basic concepts only such as parameterization and combination. "Rename" when the data type is defined by the concept of renaming. Finally a data type is of "actual" kind if it is obtained by actualization. This information can be retrieved by looking up the tree structure of the data type associated with the node.

When the kind is "normal", the list of equations of a data type is equal to the list of equations introduced in its text, plus the union of all lists of equations associated with the data types imported by that data type. For equations belonging to the imported data types the mapping φ is kept intact, while for equations introduced in the data type it is set to "nil". For example the list of equations corresponding to the data type *queue(data)* in the above specification of Figure 4.3 is equal to the set of equations introduced in its text, because the data type *data* has no equations, (for the definition of *queue(data)* and *data* see Chapter 2). The signature of each equation is also calculated e.g. $\Sigma(first(new) = d0) = \{first, new, d0, queue, data\}$, and $\Sigma(rest(new) = new) = \{rest, new, queue\}$. All mappings are set to "nil".

In case of a "rename", the process of computation consists of duplicating the list of equations at the node associated with the renamed data type and update the mapping for each equation in the list. The update of the mapping is done by composing the existing mapping with the one introduced by the renaming restricted to the signature of each equation. For an equation eq we denote its restricted mapping by $\varphi(eq)$. The restriction of the mappings will be justified later in this section when we discuss the process of eliminating duplicated equations. For example, the list of equations for the *connect*(*objects*) is equal to the list of equations of *queue*(*data*) except that the mapping of each equation is updated e.g. the mapping of the equation $first(new) = d0$ is changed to $\varphi_1(first(new) = d0) = \{(channel, queue), (object, data), (d0, d0), (new, new)\}$, and for $rest(new) = new$ becomes $\varphi_2(rest(new) = new) = \{(rest, rest), (channel, queue), (new, new)\}$.

Note that the duplication of the list of equations of the renamed type is necessary before updating the mappings. If we do not duplicate the list of equations, then the mapping of each equation in the renamed type is changed. Thus any other usage of this type will be incorrect, so the duplication avoids such problem.

When the kind is "actual", we first duplicate the equations of the actualized type and update their mapping by composing the mapping of each equation with the mapping introduced by the actualizing function restricted to the signature of the equation. Then we copy all the equations of all actualizing types and compute the union of all these equations and the ones of the actualized type after updating the mapping. For example the list of equations for the *nat_num_queue* is equal to the list of equations of *queue*(*data*) with the updated mappings, union the list of equations of *nat_num* which is empty. In this case, the equation $first(new) = d0$ has a mapping $\varphi'_1(first(new) = d0) = \{(first, first), (new, new), (zero, d0), (queue, queue), (nat, data)\}$. The duplication of the list of equations is also necessary

in this case. Suppose that the duplication has not been made in either case, then the equation $first(new) = d0$ will have the same mapping within the nodes of the two associated data types $\varphi_1''(first(new) = d0) = \{(receive, first), (new, new), (zero, d0), (queue, queue), (objects, data)\}$ which is semanticaly incorrect. This error is due to the fact that in the definition of *nat_num_queue* we do not have for example a replacement pair *receive* for *first* but the mapping associated with the above equation has this pair. Thus the translation of the data type is incompatible with its specification.

As mentioned earlier in this section, duplication of equations is eliminated first within each node of the ordered list. We consider two equations eq_i and eq_j to be equal if the following holds : the two equations have the same tree structure and the mappings $\varphi_i(eq_i)$ and $\varphi_j(eq_j)$ are equal. This justifies the choice of restricting the domain of the mapping to the signature of the equation, because two equations with the same tree structure can be equal in spite of the fact that their mappings are different. It is possible that two different mappings give the same result when applied to the same equation. This can happen when the difference of the two mapping domains is disjoint from the signature of the equation. For example suppose that we have a data type definition t_1 that imports a data type definition t_0 and that t_1 has been renamed in two occasions to obtain t_2 and t_3. Suppose also that the renaming functions used to obtain t_2 and t_3 are different and they do not apply to any operations or sort of t_0. Then the equations of t_0 belong to two different data types and when combining these two types we should eliminate their duplication. This is the purpose of restricting the mapping to the signature of an equation. For the above example the mapping of each equation coming from t_0 will be equal to the mapping of the equation as it is in t_0 so during the union of the lists of equations

of t_2 and t_3 only one occurrence of the equations of t_0 will be present because the two occurrences are considered equal.

4.2.2.2 Building the New Type

This step of the translation process consists of building a new type composed of two parts: the signature, and a list of equations. The signature is equal to the union of the signatures of all non-formal user-defined data types in a LOTOS specification. The list of equations is the union of all the lists of equations corresponding to non-formal user-defined data types in the ordered list. Formal data type definitions are not included in the resulting type, since formal sorts can appear in LOTOS specifications, but they are actualized before we get to this stage. Indeed an expression that has a sort which is formal is considered as semanticly incorrect, and this is detected during the static semantic analysis of the LOTOS specification as described in [BRD 86]. Since as mentioned earlier in §4.2.1, the internal representation of the data types is error free, then we can assume that expressions of formal sorts should not exists.

The need of constructing another list of equations instead of directly outputting the equations derives from the fact that we have to eliminate duplicated equations belonging to different nodes in the ordered list. Consider the example of last section. The equations of t_0 belong to the node associated with t_2 and the node associated with t_3. If the contents of these two nodes are to be outputted, then we must eliminate duplicated equations first, and this is done by calculating the union of the lists of equations of these two nodes prior to the output.

4.2.2.3 Outputting the Constructed Data Type

At this stage, the new type is represented by its signature and its list of equations constructed in the previous stage (§4.2.2.2). The output of the signature is straightforward by outputting each signature associated with each type. Instead, the output of the list of equations involves the application of the mappings to the associated equations. This is has to be done because in the last step of the translation, when a type is used by another, its list of equations is added to the list of the other after updating the mapping of each equation, but the tree structures of those equations are left intact. Thus before outputing an equation it is necessary to apply to it its associated mapping in order to obtain its real text. For example, for the equation represented by $first(new) = d0$ in *nat_num_queue* the associated mapping is $\varphi_1'(first(new) = d0) = \{(first, first),\ (new, new),\ (zero, d0),\ (queue, queue),\ (nat, data)\}$ and the equation is outputted as $first(zero) = zero$. Figure 4.4 shows the result of translating the data types definitions given in Figure 4.3. The form of the resulting data type specification is discussed in the next section (§4.2.3).

---

sort *new* ==> *channel*.
sort *channel* ==> *rest*(*channel*).
sort *channel* ==> *send*(*objects*, *channel*).
sort *objects* ==> *receive*(*channel*).
sort *objects* ==> *d0*.
sort *nat* ==> *zero*.
sort *nat* ==> *succ*(*nat*).
sort *nat* ==> *plus*(*nat*, *nat*).

$\| = - => rest(new) = - = new.$
$\| = - => receive(new) = - = d0.$
...
$\| = - => plus(X, zero) = - = X.$
$\| = - => plus(X, succ(Y)) = - = succ(plus(X, Y)).$

Figure 4.4 The Translation of ADTs of Figure 4.3

4.2.3 Choosing the Form of Output

The purpose of the **translator** is not only to produce a low-level specification that can be easily interpreted. It is also required to output this specification in a consistent form that can be accepted by the PROLOG system without introducing any ambiguities. The reason for this is the fact that the output of the **translator** is to be accepted by the **ADT interpreter** or the **validator** which were implemented in PROLOG. In this section, we discuss the choices made for the form of output of the resulting data type.

If we look at the specification of Figure 4.3, we see that some symbols in the data type specifications are predefined operators in PROLOG so these symbols must be renamed to avoid ambiguities. For instance the equal sign that separates the two sides of an equation is known to the PROLOG system as the unification operator, and its use in a different context as above will cause problems. Therefore, the first process of defining the data type output form is to rename all the symbols in the data type syntax that may cause ambiguities. Table 4.1 gives this list of renaming. These new symbols must be declared as new PROLOG operators by the **ADT interpreter** and the **validator**, and this is done using the PROLOG *op* predicate.

Now that the problem of ambiguity is resolved by renaming common symbols as described above, we must worry about representing the signature in a form that the sort calculation and checking is as efficient as possible. For this representation of the signature, we use the fine-grained approach used in [BMS 80], consisting of relating each sort with its constructors. For this purpose, we introduce a new PROLOG operator to denote a sort declaration. This new operator is called *sort*. For example the following declaration $or : bool, bool -> bool$ of the boolean operator *or*, will be represented internally as: $sort\ bool\ ==>\ or(bool, bool)$. This is to mean that a

Original Symbol	New Symbol
-->	=>
=	=-=
==>	=-=>
;	.

Table 4.1 Original Symbols and Their Renaming

term of sort *bool* may be obtained by using the *or* operator and two other terms with sort *bool*. This notation allows us to use the PROLOG unification mechanism to recursively calculate the sort of a term, and check for its sort correctness.

The last step is to find a common form to represent both simple equations and conditional equations to avoid different manipulations for each kind. Since the syntactic difference between these two kinds of equations is the existence of the premises for the latter and not for the former, we choose to represent the premises as a PROLOG list such that in the case of a simple equation, we simply write its premiss as an empty list. This avoids the need to distinguish between simple and conditional equations. As an example consider the equation $plus(x, zero) = x$; this will be represented internally as $[] = - => plus(x, zero) = - = x$.. This representation does not alter the semantics of simple equations since simple equations can be viewed as conditional equations without premises (see §3.6.1).

4.3 Basic Modules

In this section we describe the modules that are common to the validator and the ADT interpreter. §4.3.1 describes the knowledge base module. §4.3.2 describes the rewriting engine module. While §4.3.3 describes the tracer module.

4.3.1 Knowledge Base

The knowledge base module consists of four modules. The first module is immutable and contains the signature of the data type being interpreted. The second module represents the equations introduced by the data type definition. This module is mutable so that equations can be added to or retracted from it when used by the validator. The third module represents the term rewriting system obtained from completing the set of equations or only by ordering it. This module is also mutable. Finally, the last module is immutable and contains the precedence of the operators. This module is used by the validator when testing for completeness, or when testing for termination only.

Each of the above modules consists of a set of PROLOG facts with the same predicate name. Thus, these modules are really PROLOG packets in the sense of our definition of a packet.

4.3.2 The Rewriting Engine

This module implements the mathematical notion of conditional TRS. It makes use of the rewriting rules maintained in the knowledge base module.

We have implemented two strategies (there are many other strategies) to rewrite a term. One is the "leftmost-outermost" strategy, while the other is a "leftmost-innermost" strategy. The user is asked to select one of these two strategies. The

default is the "leftmost-innermost" strategy, that was found to be faster than the "leftmost-outermost" one in several cases.

To rewrite a term t using the "leftmost-outermost" strategy, we proceed as follows: we attempt to rewrite t at its root (as defined in §3.2 on page 22) using each rule in the conditional TRS. If this is unsuccessful, we then attempt to rewrite each of the immediate subterms of t using each rewrite rule, and so on.

In the case of a "leftmost-innermost" strategy, a term t is rewritten as follows: recursively rewrite the subterms of t starting from the leftmost one using each rule in the conditional TRS, and so on up to the root.

To improve the performance, all the reductions accomplished during the rewriting of a term are memorized, in order to avoid reducing again a term which was previously encountered and completely reduced.

Operations are available for reducing a term once with respect to the conditional TRS, and for computing the normal form of a term. A means is provided for terminating an excessive number of rewrites during a normal form computation.

Two kinds of rewriting are supported by this module, sorted rewriting and non-sorted rewriting. For the former, prior to the reduction of a term, its sort is calculated relative to the signature existing in the **knowledge base**, and checked to be correct. This kind of rewriting is used for experimentation purpose, or when the **ADT interpreter** is used within the LOTOS interpreter and the term to be reduced is the one introduced by the user at the interaction points (see [BRD 86] for the usage of the LOTOS interpreter). The other kind of rewriting is used by the **validator**, and by the **ADT interpreter** for rewriting terms that belong to the LOTOS specification, which are known to be of correct sort.

4.3.3 The Tracer

There are some interactions with the user that are displayed at low-level interfacing. Among others are the information messages printed by the conditional Knuth-Bendix procedure implementation, the messages printed by the sort checker, and the ones printed by the rewriting engine. For these situations, the **tracer** module is provided. The tracer provides a different procedure for each possible type of output message produced by the **ADT interpreter**, or by the **validator**.

4.4 The Validator and its Modules

This section describes the **validator** which is mainly consists of an implementation of an improved version of Kaplan's completion procedure described in §3.6.4. We first describe the sub-modules that implement various general mechanisms used by the resulting Conditional Knuth-Bendix procedure, such as *unification*, *ordering*, etc. We then describe the improvements implemented on Kaplan's procedure. Finally we discuss the organization of the resulting procedure.

Figure 4.5 shows the **validator** modules dependency diagram. There is an arc from a module A to another module B, if A directly uses B in its implementation. The remaning of this section is organized as follows: §4.4.1 describes the **unify** and **superpose** sub-modules. §4.4.2 describes the **solve** sub-module. §4.4.3 describes the **rpo** sub-module. §4.4.4 presents the improvements made on the completion procedure. Finally §4.4.5 describes the conditional Knuth-Bendix completion procedure organization.

4.4.1 Unify and Superpose

The procedure **unify** takes two terms and returns the most general unifier of those terms. The unification algorithm is an extended version of the PROLOG unification algorithm that includes the "occur-check" test to avoid infinite unification

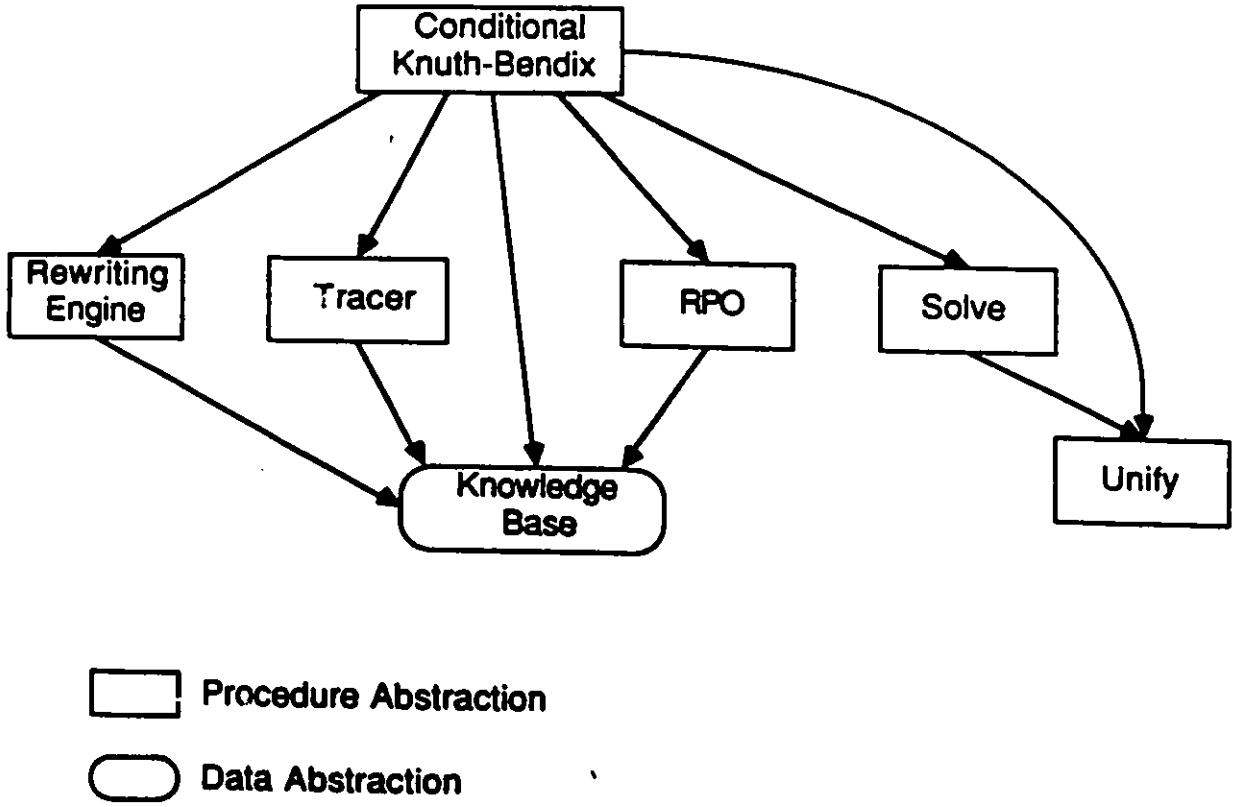

Figure 4.5 The Validator Modules Dependency Diagram

of the two terms. An example of infinite unification is the case where we try to unify the two terms x and $f(x)$, and we obtain an infinite term $f(f(...f(f(x))))$.

The procedure **superpose** takes two conditional rewrite rules, computes the superpositions associated with each overlap between the left-hand sides of their conclusions , and returns all the CCP's resulting from those superpositions. This procedure is the heart of the confluence test in the conditional Knuth-Bendix com-

pletion procedure implementation.

4.4.2 Solve

This module implements the $unify_{\mathcal{R}_c}$ procedure described in section §3.6.3 and shown in Figure 3.4. The procedure **solve** takes two terms, succeeds if the two terms are unifiable relative to the current conditional term rewriting system and fails otherwise. This procedure is used by the conditional Knuth-Bendix procedure to check for feasible CCPs.

4.4.3 RPO

This module implements the Recursive Path Ordering (RPO) machanism introduced by Dershowitz [DER 82] and described in §3.7.2 to compare two terms using the precedence of the operators used to build the two terms. This module is used by the conditional Knuth-Bendix implementation, and by the **validator** to check for termination of the conditional TRS.

This module is composed of one procedure that changes its function depending of the environement where it is used. This procedure takes a conditional equation and attempts to order it.

When used by the conditional Knuth-Bendix implementation this procedure changes behaviour from one task to another. In the **treat CCPs** and **consider large formulas** tasks of Section 4.4.5 this procedure is quiet, and no interactions with the user are performed, while when it is used in the **consider postponed formulas**, the procedure makes some interactions with the user. In the former case this procedure just returns the result of comparing the terms of the conditional equation. In the latter case the procedure may return the comparison, or may

inform the completion procedure that the user wishes to postpone the formula, interrupt the completion process, or hand-order the formula.

This procedure can be used by the validator to check only for the termination of the conditional TRS being used. In this case the procedure also makes some interactions with the user as described above.

4.4.4 Improving the Conditional Knuth-Bendix Procedure

As originally formulated by Kaplan (see §3.6.4), the conditional Knuth-Bendix completion procedure is used to transform a set of conditional equations $\mathcal{E}_c$ into a conditional TRS $\mathcal{R}_c$, such that $\mathcal{R}_c$ is complete and $=_{\mathcal{E}_c}$ equal $=_{\mathcal{R}_c}$.

The procedure is shown in Figure 3.5. This formulation was chosen by its author in view of considerations of simplicity of exposition and ease of proof, rather than efficiency. Therefore, in our implementation, we had to deal with two problems with this formulation:

1. it is inefficient, and
2. it fails whenever an unorderable formula is generated.

In the above and what follows, by a *formula* we mean an equation or a conditional equation.

In this section, we describe the improvements implemented on Kaplan's procedure to deal with these issues. As a partial solution to (1) above, we incorporate improved mechanisms for generating CCPs and normalizing the rewriting system. For (2), we use a fine-grained approach to postponing formulas that are currently unorderable. These improvements give us a completion procedure which is potentially faster and which halts with "failure" in fewer cases.

The major improvements of our version over Kaplan's are:

1. Our procedure generates the contextual critical pairs between any two rules only once, whereas the original procedure begins again to look for CCPs among all rules during each iteration through the main loop. The speed-up attained in our formulation can be significant, since the unifications required in computing CCPs and checking for feasible ones can be time-consuming.

2. Our procedure does not reorder conditional rewrite rules for which the premises and/or the right-hand side of the conclusion is rewritten during normalization but the left-hand side is left intact. This re-ordering is unnecessary because such rules will still be ordered under the reduction ordering. For instance, let $\wedge_{i=1}^{n} p_i = q_i \Rightarrow \lambda \rightarrow \rho$ be an arbitrary rule in $\mathcal{R}_c$, which satisfies the conditions: i) $\lambda \succ p_i$ ii) $\lambda \succ_{\text{rpo}} q_i$ iii) $\lambda \succ_{\text{rpo}} \rho$. If ρ is reduced to ρ', then $\rho \succ_{\text{rpo}} \rho'$ (see §3.7.2) We know (see §3.7.2) that the ordering $\succ_{\text{rpo}}$ is transitive hence $\lambda \succ \rho'$, and the rule $\wedge_{i=1}^{n} p_i = q_i \Rightarrow \lambda \rightarrow \rho'$ satisfies conditions i), ii), and iii) then this rule is already ordered and should be added to $\mathcal{R}_c$. The same argument could be used in case where the premises are rewritten during normalization and the right-hand side of the rule is left intact.

3. Our procedure implementation does not fail when an unorderable formula is found. The formula may be postponed. Postponement might allow the formula to be reduced, to disappear, or to be ordered later.

4. Our implementation automatically postpones consideration of large formulas.

5. Our implementation computes smaller contextual critical pairs first, which can speed-up the completion process (§4.4.4.1).

§4.4.4.1 describes the techniques used to compute CCPs between any two rules only once, and the computation of small CCPs. §4.4.4.2 describes formulas postponement. §4.4.4.3 describes the use of user interaction in case of an unorderable

formula. §4.4.4.4 discuses the normalization of the rewriting system upon addition of a new rule.

4.4.4.1 Computing Small Contextual Critical Pairs

The scheme for computing CCPs can be characterized as follows: Maintain the rewriting system as a PROLOG packet. Each fact in this packet consists of a rewrite rule and its integer label. Each rewrite rule that gets added to the packet is initially unmarked. In the critical pair computation step, an unmarked rule, say $\wedge_{i=1}^{n} p_i = q_i \Rightarrow \lambda \rightarrow \rho$, is chosen and all contextual critical pairs between it and every rule with label less than its label, including itself in the packet, are computed. Then this rule is marked (the marking of a rule is by memorizing its associated label). In this way, each distinct pair of rewrite rules is used only once. This idea is due to G. Huet who applies it to the non-conditional case. We have found that the same idea is applicable in the conditional case, so we incorporate this technique of marking rules to the conditional Knuth-Bendix completion procedure. This improves the efficiency of the original procedure of §3.6.3. Indeed the unification required in computing CCPs can be time-consuming, thus computing CCPs in this manner may speed-up the completion process.

Another technique which is known in the equational case involves computing small critical pairs first.

In [KNU 70], the authors note that small pairs of rewrite rules are more likely to lead to small critical pairs. Small critical pairs are useful because they take less time to generate and tend to lead to more general rules than do larger critical pairs. It is often the case that these rules reduce larger rules and equations, thus reducing the number of larger critical pairs that need to be generated. For example, consider

the following rewriting system:

$$(1) \quad plus(x, plus(0, y)) \rightarrow plus(x, y).$$
$$(2) \quad plus(0, plus(i(i(x)), y)) \rightarrow plus(x, y).$$
$$(3) \quad plus(0, i(i(x))) \rightarrow x.$$

According to the above discussion, it seems more appropriate to generate critical pairs between (3) and (1) first. Indeed, such choice is the most appropriate. The critical pair $\langle plus(x, i(i(y))), plus(x, y)\rangle$, obtained by superposing rule (3) against (1), may be oriented to obtain rule (5) $plus(x, i(i(y))) \rightarrow plus(x, y)$. This rule causes rule (3) to be replaced by rule (6) $plus(0, x) \rightarrow x$, which in turn causes rule (1) to disappear, and causes rule (2) to be replaced by rule (7) $plus(i(i(x)), y) \rightarrow plus(x, y)$.

One of the schemes to calculate small critical pairs is to maintain the packet of rewrite rules so that it is sorted by size, and if critical pairs with chosen rules are calculated with rules above it in order from the top of the packet down to itself, the marking scheme will ensure that critical pairs are always calculated starting with smallest pairs that have not yet been considered.

We also found the idea of computing small critical pairs helpful to efficiently implement the conditional completion procedure. Small contextual critical pairs can be obtained by using small pairs of conditional rewrite rules.

Our implementation uses a strategy for choosing pairs of rules, which does not maintain the packet so that it is sorted by size, because in PROLOG it is difficult (and especially very inefficient) to sort a set of facts by size. But in fact we do not want to choose the smallest pair of rules first but the smallest pairs first. This has been implemented by maintaining a *Bound of Complexity BC*, and a rule of

complexity C is added to the packet according to the following criterion:

if $C \leq BC$ **then** the rule is added at the top of the packet of rules.

if $C > BC$ **then** the rule is added at the bottom of the packet of rules.

The bound of complexity BC is set to be equal to the largest size of the formulas introduced by the user.

4.4.4.2 Postponing Formulas

In its original formulation, Kaplan's conditional Knuth-Bendix completion procedure halts with "failure" as soon as a rewrite rule is found that cannot be ordered. However, as noted in [DER 83b], one can postpone consideration of unorderable equations, and abort only if there are no orderable ones. This has been done in systems dealing with non-conditional equations only (see [LES 84]). This idea of postponing equations is incorporated in our implementation which deals with conditional equations. There are several good reasons for doing this. Unorderable formulas might later become orderable, or disappear entirely, when reduced by a new rewrite rule. The user may decide to hand-order it later (if it can be viewed as a conditional rule as discussed in §3.6).

We mentioned early in the last section the usefulness of generating small contextual critical pairs first. For similar reasons, it is advantageous to consider small formulas first. Thus, our implementation postpones large formulas, in addition to the unorderable ones.

Our implementation partitions the formulas into three packets. The formulas that have not yet been treated, are in the **feasible critical pairs packet.** The postponed unorderable formulas are maintained in the **postponed packet.** The large postponed formulas are in the **large packet.**

The postponed packet contains formulas that are referred to as *incompatible*, or *unoriented* formulas.

An *incompatible* formula is one that cannot be viewed as a conditional rewrite rule in either direction, as discussed in §3.6.

An *unoriented* formula is one that can be viewed as a conditional rewrite rule, but is unorderable at the present time.

These formulas are being postponed, rather than hand-ordered, because the user hopes that a later rule will reduce the formula to make it orderable.

Our implementation does not look at large formulas until other formulas have been ordered or postponed, and all contextual critical pairs have been computed. The number of symbols in each large formula is greater or equal to γ, a value maintained by the implementation in such way that none of the formulas in the initial set introduced by the user is considered as large formula. The size of all other postponed formulas is less than γ.

4.4.4.3 User Interaction in Case of Unorderable Formulas

When dealing with the orientation of formulas, user assistance is sometimes needed. When we encounter a formula that the ordering algorithm is unable to order, this formula is shown to the user. The user is then asked to choose an action from a subset of choices shown in Figure 4.6.

---

(1) Postpone the formula for the time being.
(2) Accept the formula as a rewrite rule in the direction shown.
(3) Accept the formula as a rewrite rule in the reverse direction.
(4) Interrupt the conditional Knuth-Bendix.

Figure 4.6 User Actions in Case of Unorderable Formula

If (1) is chosen, the formula is added to the postponed set. If (2) or (3) are chosen, the formula is added to the rewriting system and the completion procedure continues its process (these last two choices are only allowed if the equation can be viewed as a rewrite rule in the selected direction as discussed in §3.6).

If action (4) is chosen, the current state of the completion procedure is recorded, so that it can later be resumed from this point, and we return to the system level.

4.4.4.4 Normalizing the Rewriting System

During the completion process, the addition of a new rule to the current system may alter the state of existing rules. Indeed some rules may be reduced or may disappear when rewritten using the new rules, so it is more effective to normalize the rewriting system each time a new rule is added to it.

The normalization process consists of keeping all the rewrite rules in normal form relative to the current rewriting system. This is has the advantages that the resulting rewriting system is minimal, and the completion process is expedited. The reduction of rewrite rules to normal forms will reduce the size of rules, thus the computed CCPs will be smaller than the ones calculated if the rules are not kept in normal forms. However rewriting some rules to normal forms may violate their orientation, so we may need to reorient them. The description of the normalization process is given hereafter.

The completion procedure does not "normalize" the entire rewriting system each time a rewrite rule is added. Rather, it uses the fact that the rewriting system is completely normalized prior to adding an additional rewrite rule, and only those rules for which the premises and/or the left or right-hand side of their conclusion can be rewritten by the new rule must be re-normalized once the rule has been

added. Furthermore, the procedure does not reorder rewrite rules for which the right-hand side and/or premises are rewritten by the new rule but the left-hand side is left intact. This reordering is unnecessary because such rules will still be ordered under the reduction ordering.

4.4.5 The Completion Procedure Organization

The conditional Knuth-Bendix completion procedure implementation is organized into different modules, and each module is assigned a specific task. Each module has a priority to be executed upon another. These priorities are established once for all during the implementation stage. The choice of priorities is done by taking into consideration the fact that a module with a task that is more "effective" in order to expedite the completion process, is to be executed before another whose task is less "effective". For instance the computation of feasible CCPs is a less "effective" task than treating feasible contextual critical pairs. Indeed, feasible CCPs are expensive to compute, and we hope that by first treating as many feasible CCPs as possible into rewrite rules, many rules and feasible CCPs will disappear when rewritten with the new rules. Thus fewer feasible CCPs will need to be computed.

Figure 4.7, shows the structure of the procedure implementation. There is an arc from a module to a packet, if the module uses this packet in its implementation.

Here is a description of each task performed by the procedure implementation in decreasing order of effectiveness.

1. **Treating Feasible CCPs** : remove a formula from the feasible critical pair packet, and reduce it to normal form with respect to the current conditional rewriting system. If the resulting formula is large, move it to the packet of large formulas. Otherwise attempt to order it. If the formula becomes a rule,

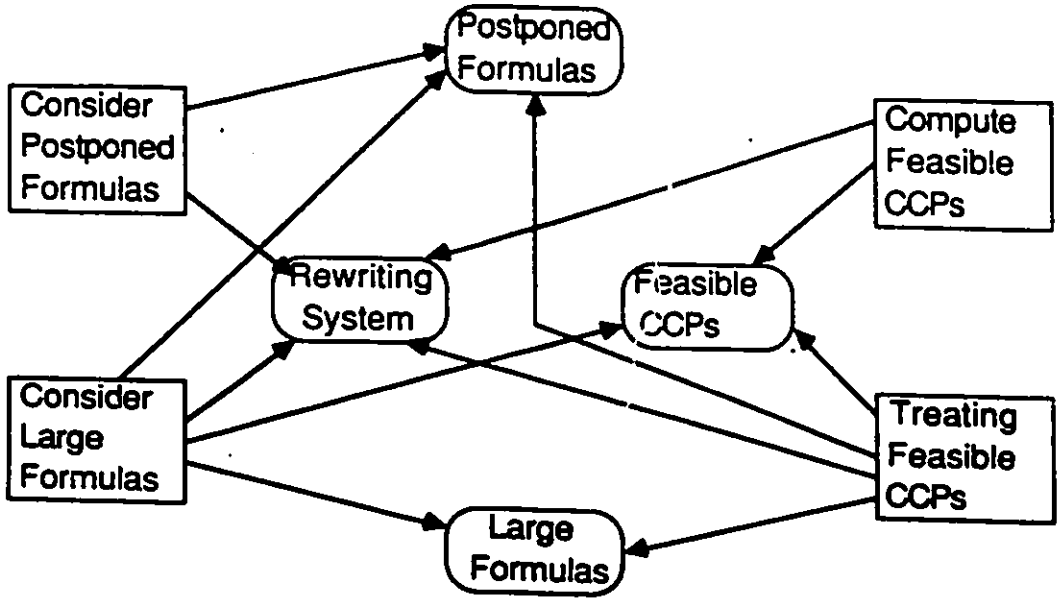

Figure 4.7 Structure of the Conditional Procedure Implementation

normalize the rewriting system as described in §4.4.4.4, and add this rule to the rewriting system. Any rule that becomes a formula as a result of normalization gets added to the feasible critical pair packet. Otherwise put the formula into the postponed packet. Repeat until the set of feasible critical pairs is empty.

2. **Consider Postponed Formulas** : remove a formula from the postponed packet, and reduce it to normal form with respect to the current rewriting system. Ask the user to choose one of the actions described in §4.4.4.3. Repeat until a new rewrite rule has been generated or all postponed formulas have again been postponed.

3. **Compute Feasible CCPs** : select an unmarked rule (relative to our scheme

described in §4.4.4.1), mark it, compute feasible CPP's between it and all other rules with label less than its label, including itself, and add the feasible CCP to the set of feasible critical pairs. If no feasible CCPs are generated, then repeat. If there are no unmarked rules, do nothing.

4. **Consider Large Formulas** : remove a formula from the set of large formulas. Process the formula in the same manner as for feasible CCPs in task 1. Repeat until a new rewrite rule has been generated, or there are no more large formulas.

Figure 4.8, shows a skeleton pseudo-code of the completion procedure implementation. The **repeat** means "go to the first statement of the smallest enclosing loop".

Initialize
While there are any formulas **do**
  Treat Feasible CCP
  if there are any untreated CCP's **repeat endif**
  Consider Postponed Formulas
  if there is any new rule **repeat endif**
  Compute Feasible CCP's
  if there are any new Feasible CCP's **repeat endif**
  Consider Large Formulas
  if there is any new rule **repeat endif**
endwhile

Figure 4.8 The Procedure Implementation Skeleton Pseudo-Code

The initialization phase consists of initializing the rewriting system to the empty set. Further, the feasible critical pair packet is initialized to the set of conditional equations introduced by the user for completion. The bound of complexity (§4.4.4.1) is also calculated in this phase. The tasks are performed in decreasing order of effectiveness, directely reflecting the assumptions made above. One hopes that the more effective tasks will reduce the work required of less effective tasks; e.g. adding new rules to the system may cause an unoriented formula to disappear after reduction to normal form.

4.5 The ADT interpreter and its Modules

This section gives an overview of the major modules of the ADT interpreter Figure 4.9 shows a module dependency diagram for most of the procedures we discuss here. There is an arc from a module A to another module B, if A directly uses B in its implementation.

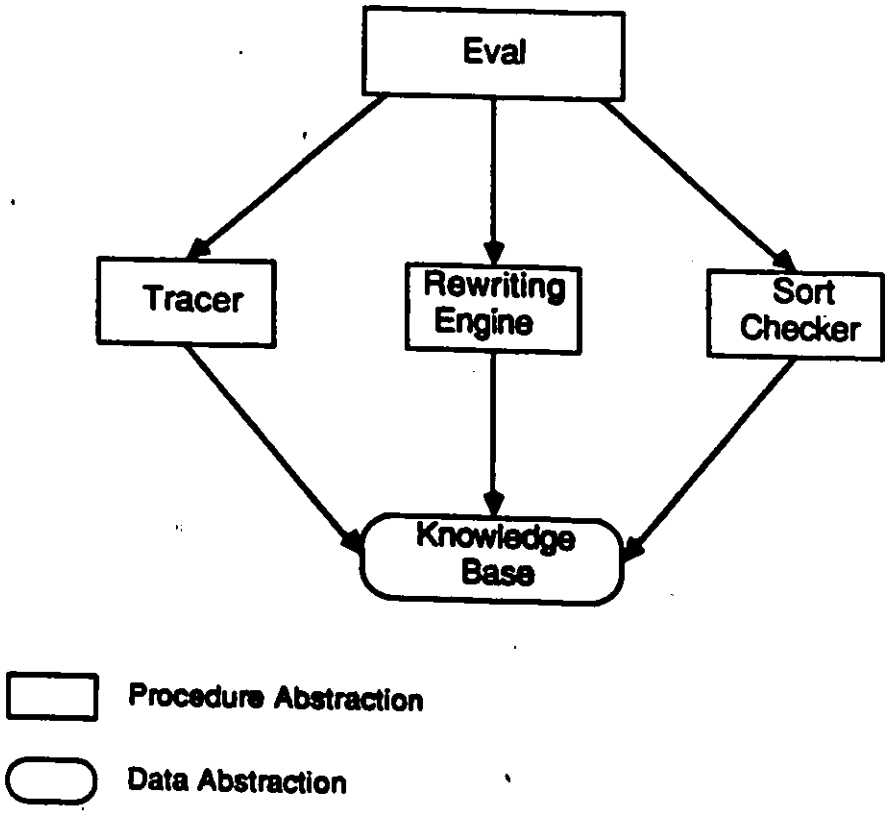

Figure 4.9 The ADT Interpreter Module Dependency Diagram

4.5.1 The Sort Checker

This module is used by the rewriting engine when sort checking is necessary prior to rewriting (see §4.3.2). The module is composed of one procedure that calculates the sort of a term with respect to the signature existing in the knowledge base. The sort calculation strategy is "leftmost-innermost". In other words to calculate the sort of a term t, we start from the leftmost subterm and we calculate its sort recursively. When a sort error is detected by the procedure during the calculation process, the error is signalled to the user with the expected sort at that occurrence. During the calculation process, well-sorted terms are memorized with their calculated sort. This improves the performance of the sort checker procedure.

As discussed in [BRD 86] the problem of operator overloading in a specification is resolved by renaming the specification in such a way that we obtain an equivalent specification with the same semantics and having the unique naming property. Therefore, the ADT interpreter will never find overloaded operators. However, we found it worthwhile to implement operator overloading. This is has been done by backtracking which is very natural in PROLOG. An example of this is given in Appendix A.

4.5.2 Rewriting with Permutative Rules

According to what we have seen so far, the rules that can be used by the rewriting engine must be simplifying rules, in other words the left-hand side of the conclusion in a rule must be (in some sence) simpler than the right-hand side, in order to assure the termination of rewriting. This forbids the use of rules that do not satisfy the above condition (we will call such rules "permutative" rules). Examples of these rules are rules expressing commutativity. However the use of these rules is sometimes indispensable to reduce a term, because a term that is not reducible

using simplifying rules can become reducible after having been transformed using one or more permutative rule. For example, consider the following system:

$$1 : [] = - => \ plus(x, zero) \ ==> \ x.$$

$$2 : [] = - => \ plus(x, i(x)) \ ==> \ zero.$$

$$3 : [] = - => \ plus(x, y) \ = - = \ plus(y, x).$$

$$4 : [] = - => \ plus(plus(x, y), z) \ = - = \ plus(x, plus(y, z)).$$

In this system, permutative rules are distinguished from simplifying rules by the predicate name used to denote a rewriting rule. When the predicate is "= - =", the rule is known to be permutative, and when it is "==>", the rule is known to be simplifying. For example, the term $t = plus(plus(x, y), i(x))$ cannot be reduced by rule (1) or (2), but can becomes reducible after transformation using rule (3) and (4) as shown below:

$$plus(plus(x, y), i(x)) \ - -3 - - > \ plus(plus(y, x), i(x))$$

$$- - 4 - - > \ plus(y, plus(x, i(x))) \ - -2 - - > \ plus(y, zero)$$

$$- - 1 - - > \ y.$$

To allow the usage of permutative rules we have extended the rewriting algorithm. In order to take advantage of this extension, we have to put the rewriting engine in the "permutative-mode" using the *permut-on* command. In the normal mode, permutative rules are ignored by the rewriting engine.

To rewrite a term in the "permutative-mode" the following strategy is followed:

1: Permutative rules have the least priority. This means that we only use them if none of the simplifying rules is applicable.

2: Permutative rules are used in either direction.

3: After each rewriting step that used a permutative rule, we try again to reduce the term using simplifiying rules.

4: If it is not possible yet to reduce the term using simplifiying rules , we perform another permutative rewrite step. While it is not possible to rewrite the term by the symplifying rules, each intermediate step is memorized to avoid infinite loops.

5: The process is stopped when it is no longer possible to perform a permutative rewrite step without obtaining a term that has already been obtained earlier during the rewriting.

An example of usage of the **rewriting engine** in the "permutative-mode" is given in Appendix A.

CHAPTER 5

SVELDA's User Interface

5.1 Introduction

SVELDA was designed and implemented as a part of the LOTOS interpreter as shown in Figure 1.1. It can be used not only within the LOTOS interpreter, but also by itself for formal theorem proving and experimentation.

SVELDA supports equational theorem proving, as well as direct access to basic rewriting, unification, sort checking, ordering, and superposition operations. Formal theorem proving is accomplished with the the conditional Knuth-Bendix implementation described in Chapter 4 (see §4.4.4 and §4.4.5) In addition, individual terms can be rewritten and unified, sorts of terms can be calculated and checked, and CCPs can be calculated for individual pairs of rewrite rules, for experimentation purposes.

This chapter describes the features provided by the SVELDA user interface.

5.2 The User Interface

SVELDA uses the PROLOG environment to communicate with the user. To start SVELDA the user types while he is in the UNIX environment the command svelda. Typing this command causes the PROLOG system to enter the PROLOG environment, and consult all the programs consisting of the submodules of SVELDA. So at this point one can communicate with the ADT interpreter and the validator using the commands of SVELDA. Commands should be typed in lower case, ending with a period. SVELDA accepts only prefix notation for terms, and prefix or infix notations for equations, conditional equations and rewrite rules. The

help command provides on-line documentation for each command, plus additional information about the use of SVELDA.

The commands supported by SVELDA fall into the following categories:

- Invoking the conditional Knuth-Bendix and theorem proving.
- Directly accessing rewriting, unification, sort calculation and checking primitives.
- Orienting a set of formulas without testing the confluence.
- Handling the input, output, display, and deletion of the rules and formulas manipulated by the ADT interpreter, and the validator
- Saving and restricting terminal input/output.

In what follows, we present an overview of these capabilities.

5.2.1 Specification

The user's current data type specification may be read from, and written to, disk files and the user's terminal, using the **readf** command. This command takes the name of the input file as its argument. Individual rules and formulas may be deleted from the specification. This can be done using the **delete** command, where the equation, rule, or formula to be deleted is given as argument.

In addition, when the set of formulas of the data type specification has been completed by the conditional Knuth-Bendix procedure, the user may **freeze**, into a file, the entire specification state, including all the current rules, formulas and the signature of the specification. Note that the signature of a specification is not used by the completion process, but is used during sort calculation and sort

checking. Later the user may thaw the frozen specification. **freeze** and thaw are particularly useful for saving completed specifications that are of general utility, or for temporarily saving an incomplete session. These two commands take a name of a file as their argument.

5.2.2 Conditional Knuth-Bendix and Proofs

The **ckb** command invokes the conditional Knuth-Bendix procedure on the set of formulas of the current specification. The completion process can be interrupted at any user's interaction point. The user can invoke other commands, and subsequently continue the completion process.

Equational formal proofs are performed with the **prove** command. This command takes a equation as its argument, and attempts to prove that the equation is in the equational or conditional theory of the current specification. **Prove** first uses the current rewriting system to reduce the equation to normal form; if the two sides of it become equal, the theorem holds. Otherwise, if the current specification has not yet been completed, the conditional Knuth-Bendix procedure is automatically invoked (after user confirmation). If the completion process terminates successfully, the equation is again normalized. If the two sides of the equation become equal, the formula is valid in the equational or conditional theory.

5.2.3 Basic Operations

Basic rewriting primitives are invoked with the **reduce** and **normal_form** commands. Both of these commands operate on a term given by the user. The **reduce** command reduces the term (if possible) once, using an arbitrary applicable rewrite rule from the current rewriting system. The **normal_form** command computes the normal form of the term with respect to the current rewriting system. If the

term gets rewritten an inordinately large number of times and no normal form has yet been found, the ADT interpreter assumes that rewriting will probably not terminate. In this case the normal form computation stops, and the user is shown the last several intermediate reduced forms to help in identifying the reason of the non-termination.

The **unify** and **critical_pairs** commands permit access to the primitive operations used by the conditional Knuth-Bendix procedure. The **unify** command accepts two terms as arguments, and displays their unification, or indicates that the two terms cannot be unified. The **critical_pairs** command displays all the feasible CCPs, if any arise by superposing two rewrite rules given by the user. These two commands are of experimentation interest.

The **sort_of** command is used to calculate the sort of a term given by the user, and to check for its correctness. The **sort_of** command accepts a term as its argument, and calculates its sort with respect to the signature of the current specification. If a sort error is detected by the ADT interpreter, the sort computation stops, showing the user the position of the error and its kind.

5.2.4 Orienting the Set of Formulas of a Specification

When the user is interested only in the termination of the rewriting system that can be produced by orienting the set of formulas of the specification, the **orient** command can be used for this purpose. The **orient** command invokes the ordering module, that causes the validator to order all current formulas into rewrite rules, using the RPO ordering, without computing any critical pairs. When a formula cannot be transformed into a rule, the user is prompted to take action on it. The actions that can be taken are to hand-order the formula or to keep the formula unoriented. In this latter case the mode of rewriting is automatically set to be

"permutative", so the unoriented formulas can be used by the rewriting engine as permutative rules.

5.2.5 Rewriting in Permutative Mode

When the rewriting system is known not to terminate, the rewriting engine must be used in the permutative mode. The formulas that cause non-termination are used as permutative rules. The strategy of rewriting in permutative mode has been described in §4.5.2. Two commands are available to set and reset the mode of rewriting. The **permut_on** command causes the rewriting engine to operate in permutative mode. The **permut_off** command will reset the rewriting mode to normal, so during the rewriting unoriented formulas are ignored. In the permutative mode, the rewriting can be used at the interpretation or experimentation level, but not within the completion process. The **reduce** and **normal_form** commands can be used in the two modes of rewriting, depending on the user's choice.

5.5.6 Terminal Session

Commands in this category control the terminal session. These commands are independent of the application domain; they do not directly pertain to rewriting, sort checking, and formal theorem proving capabilities of the SVELDA.

One of these command is the **script** command that is not part of our system, but belongs to the UNIX operating system. The **script** command must be issued before the **svelda** command, and causes all terminal input/output to be sent to a predefined file named *typescript* for later viewing.

The **trace_on** command can be issued by the user in one of the following contexts of application.

When issued during the completion process, the conditional Knuth-Bendix implemetation is capable of displaying many kinds of information, including

- The next formula under consideration.
- The normal form of that formula.
- The rewrite rule that comes from that formula.
- The rewrite rules whose right-hand sides are rewritten by the new rule.
- The rewrite rules whose left-hand sides are rewritten by the new rule.
- The rewrite rules that have disappeared as a result of rewriting.

Using the **trace_on** command, in the context of proving theorems, calculating normal forms, or sort calculation, all intermediate rewriting or calculated sorts are displayed to the user. The **trace_off** command disables all displays.

CHAPTER 6

Summary and Conclusion

6.1 Summary and Contributions

In this thesis, we have introduced LOTOS "data" part concepts. These are based on the algebraic specification language ACT ONE. Concepts of term rewriting and conditional term rewriting as well as equational proofs, and conditional proofs, were introduced in a tutorial manner. We have improved the efficiency of the conditional Knuth-Bendix procedure introduced by Kaplan. We have designed and implemented a translator from ACT ONE texts used in a LOTOS specification to a low level specification language that can be easily interpreted. We have designed and implemented an ADT interpreter for these low level specifications. Finally, we have designed and implemented a validator that incorporates the conditional Knuth-Bendix procedure implementation. These three tools pertain to SVELDA.

SVELDA was designed not only to be used within the LOTOS interpreter, but also to be used as an experimentation tool. SVELDA consists of three main modules, the translator, the ADT interpreter, and the validator. The following tasks can be performed by SVELDA.

- Translating ACT ONE texts existing in a LOTOS specifications into a low-level data type specification, which is the union of all data types introduced in the LOTOS specification (translator).

- Testing the confluence and/or the termination of a conditional term rewriting system that can be produced from a set of conditional equations (validator). This consists of running the conditional Knuth-Bendix procedure, or running the ordering procedure.

- Proving equational theorems (validator).

- interpreting expressions constructed from some defined data types (ADT interpreter). This is the case when SVELDA is used within the LOTOS interpreter.

- Calculating the sort of individual terms and checking for sort correctness (ADT interpreter).

In the remaning of this chapter, we will describe the current limitations of SVELDA, and highlight some ideas for further work.

6.2 Current Limitations and Future Work

The LOTOS interpreter continues to be enhanced, both with new features and with fine tuning of existing features. We list here some of the improvements that are under consideration for SVELDA, which is (as mentioned earlier across this thesis) a part of the LOTOS interpreter.

6.2.1 Conditional Equational Term Rewriting Systems

The correctness of the conditional Knuth-Bendix procedure requires that the rewriting system terminate at each step of the procedure. This requirement disallows the use of conditional equations sets (we mean here both conditional and simple equations) that include, for example, useful permutative equation such as $plus(x, y) = plus(y, x)$.

To handle this problem in the equational case, Huet [HUE 80b] and Peterson and Stickel [PET 81] have extended the Knuth-Bendix procedure to operate on Equational Term Rewriting Systems (ETRS): an ETRS is a rewriting system, together with a set $\mathcal{E}$ of equations, that are not converted into rules. The completed rewriting system, together with $\mathcal{E}$, provides a decision procedure for the equational

theory of the equations and rules that comprise the ETRS. Huet's method requires that all rewrite rules be left-linear (for every rule, each variable appears at most once on the left-hand side). The Peterson and Stickel approach is limited to systems where $\mathcal{E}$ consists only of equations that are both left and right-linear, and where a finite and complete unification algorithm for $\mathcal{E}$ is known ("$\mathcal{E}$-unification" is the process of finding a set of maximally general substitutions for the variables of two terms , that make those two terms equal in the theory of $\mathcal{E}$).

In [JOU 83] the approach of Peterson and Stickel has been generalized by allowing non-linear equation in E. However, [JOU 83] does not propose a particular completion procedure that incorporates these new results. Neverthless Jouannaud and Kirchner [JOU 84] simplify, generalize, and extend the [JOU 83] results about ETRS. They use these new results to prove the correctness of a new completion procedure that is more powerful and more efficient than the previous methods.

We conjecture that the conditional Knuth-Bendix procedure described in Chapter 3 and improved in Chapter 4 can be extended to operate on Conditional Equational Term Rewriting System (CETRS): conditional rewriting systems together with a set of (non-oriented) conditional equations. But this needs additional investigations.

6.2.2 Inductionless Induction

The only proofs that can be performed by the validator are the ones that consist of proving equational theorems. Althought the notion of "equational theory" is useful in the context of algebraic structures, like groups, it is less useful in the context of abstract data types, where we deal with inductive theory. Huet [HUE 82] has introduced the notion of inductionless induction, and extended the Knuth-Bendix completion procedure to prove inductive theorems without the need of the

induction process. Goguen [GOG 80] has also worked on the same notion to show how to prove algebraic inductive hypotheses without induction. Their work has been limited to the framework of the equational case. However E. Paul [PAU 84] has extended these results to the case of conditional theory. We feel that these particular results naturally extend to Kaplan's formalism. Thus the conditional Knuth-Bendix completion procedure can be improved to incorporate inductive proof capabilities.

6.2.3 Simplification Ordering

Currently the only simplification ordering used to prove termination of the rewriting system is the RPO described in [DER 82], and presented in §3.7.2, which fails to orient some fomulas that seem to be orientable i.e., $plus(plus(x,y),z) = plus(x,plus(y,z))$. However some more powerful orderings have been developed during the past few years which are extensions of the RPO or are alternative solutions that contain the RPO. As an extension to the RPO, [KAM 84] has developed a new simplification ordering to deal with the status of the operators, called Recursive Path Ordering with Status (RPOS). [JOU 82a], has introduced the Recursive Decomposition Ordering (RDO) mechanism as an alternative to the RPO introduced by Dershowitz, and shown that this ordering is more powerful and contains the RPO. In Lescanne [LES 84] the result of [JOU 82a] has been extended to deal with the status of operators producing a new mechanism called Recursive Decomposition Ordering with Status (RDOS), that is more powerful than RPO, RDO, RPOS and proven to contain these three orderings.

A promising direction of further work is to incorporate both RPOS, and RDOS implementations in the ADT interpreter, so that the construction of terminating rewriting systems can be more effective: more formulas are oriented, and the user

interaction is reduced. Since the RDOS mechanism is more powerful than the RPOS, but the RPOS is more efficient than the RDOS. An efficient implementation of a module that contains these two mechanisms to prove termination is as follows: given a formula to be oriented we first try to order the formula using the RPOS mechanism. if the RPOS fails to order the formulas, the RDOS is used for assistance. In this way, the RDOS is used only when the RPOS is not powerful enough to order the formula. In this manner the ordering procedure is as efficient as possible.

As mentioned in §4.3.2, the rewriting engine memorizes already reduced terms to increase rewriting speed. This is a space-consuming approach, but it is the only alternative that we can use to replace other methods introduced in the literature, which are difficult if not impossible to efficiently implement in PROLOG.

One of these techniques is the one used in Affirm [MUS 80a]. The idea behind this technique is to use a hash table that maps operators to buckets of "pointers," where each "pointer" points to a rewrite rule in the set. The root operator on the left-hand side of each rule conclusion serves as the hash key for that rule. When reducing a term or a subterm $t = f(\ldots)$, the rewriting operation only needs to try the rules referenced by the bucket associated with f. Rules not referenced by "pointers" in that bucket will not match t.

Another method to speed up normal form computations has been developed by Plaisted [PLA 83]. He suggests associating a hash table with the rewriting system, where the hash keys are terms, and the values stored in the hash are rewrite rules. Whenever the normal form t_2 of a term t_1 is found, one adds the rule with $t_1 \rightarrow t_2$ to the hash table, under the hash key t_1. When computing the normal form of a term t_3 first hash t_3 and try to match t_3 against the left-hand side of each rewrite rule conclusion in the resulting hash bucket. If a match is found, rewrite t_3 using

that rewrite rule, and then compute the normal form of the resulting term with respect to the rewriting system. In this way, several reduction steps can often be skipped.

These powerful ideas are not incorporated in our implementation, due to the limitations of PROLOG as mentioned earlier. But we suggest that these idea of increasing the speed of rewriting should be considered in future implementations of the ADT interpreter, where the language of implementation will have some capabilities to support data structures.

6.2.4 Computing Small Critical Pairs

As discussed in §4.4.4.1, smallest CCPs are more desirable than larger ones. It is difficult, if not impossible, to determine the size of a CCP in advance (in general). However, §4.4.4.1 notes that is a good heuristic is to pick a small pair of rewrite rules with which to compute CCPs. Since CCPs are expensive to compute, it is useful to generate only a few CCPs at a time. If these can be ordered into rules, they might be reduced to eliminate larger rules reducing the number and size of CCPs that must be computed.

The marking scheme in §4.4.4.1 will always use a pair of rules that is one of the smallest pairs. A drawback though, is that many CCPs may be generated at once. We can generate fewer CCPs at once if we pick the smallest pair of rules, and only compute CCPs between those rules before attempting to order CCPs. This scheme requires more bookkeeping.

Kapur and Sivakumar [KPR 83] have extend this idea further, in the equational case, by generating critical pairs one at a time. Once the smallest pair of rewrite rules has been identified, only one critical pair (if any exists) is generated from the

pair of rules. After handling the critical pair (i.e. by ordering it into a rewrite rule and normalizing the rewriting system accordingly) if that same pair of rules is still the smallest pair, the next critical pair between these rules is generated. This idea can be easily incorporated in our procedure implementation which operates on conditional equations. But since this approach requires that the rewriting rules be kept sorted, which is very inefficient to do in PROLOG, we have chosen not to implement this feature. However we feel that this technique would be of great utility to expedite the completion process, and we hope that it will be incorporated in future implementations using more efficient programming languages.

REFERENCES

[BAX 73] L. D. Baxter, "An Efficient Unification Algorithm," Technical Report CS-73-23, Dept. of Applied Analysis and Computer Science, Univ. of Waterloo, Waterloo, Ontario, 1973.

[BER 82] J. Bergstra and J. Klop, "Conditional Rewrite Rules: Confluence and Termination," Report IW 198/82, Amsterdam 1982.

[BRD 86] J.P. Briand, M.C. Fehri, L. Logrippo, and A. Obaid, "Executing LOTOS Specifications," to appear in *Proc. of the VIth IFIP Workshop on Protocol Specification, Verification, and Testing*, Montreal, Quebec, 1986.

[BRI 85] E. Brinksma, "A Tutorial of LOTOS," *Proc. of the 5th International Symposium on Protocal Specification, Testing, and Verification*, Toulouse-Moissac, 1985.

[CAR 84] V. Carchiolo, A. Faro, and G. Scollo, "A Temporal Ordering Specification of Some Session Services," in: Communications Architectures and Protocols, *Proc. of the 1984 SIGCOMM Conf*, 107–114.

[CME 84] W. F. Clocksin, and C. S. Mellish, "Programming in PROLOG," Springer-Verlag, Berlin, 1984.

[COR 83] J. Corbin and M. Bidoit, "A Rehabilitation of Robinson's Unification Algorithm," R. E. A. Mason (Ed.), *Proc. 9th World Computer Congress*, IFIP '83, North-Holland, September 1983, pp. 909–914 .

[DER 79a] N. Dershowitz and Z. Manna, "Proving Termination with Multiset Orderings," *Communications of the ACM* 22(8):465–476, 1979.

[DER 79b] N. Dershowitz, "A Note on Simplification Orderings," *Information Processing Letters* 9:(5)212–215, 1979.

[DER 82] N. Dershowitz, "Orderings for Term-Rewriting Systems," in *Theoretical Computer Science, Vol. 17*, North-Holland, 1982, pp. 279–301.

[DER 83a] N. Dershowitz, "Computing with Rewrite Systems" *Proc. of an NSF Workshop on the Rewrite Rule Laboratory*, Sept. 6–9 1983.

[DER 83b] N. Dershowitz, "Applications of the Knuth-Bendix Completion Procedure," Technical Report ATR-83(8478)-2, Aerospace Corp., El Segundo, CA, May 1983.

[EHR 85] H. Ehrig, and B. Mahr, "Fundamentals of Algebraic Specifications 1," Springer-Verlag, Berlin 1985.

[FAY 77] M. Fay "First-Order Unification in Equational Theory," *Proc. 4th Workshop on Automated Deduction*, Austin, TX, Feb. 1977, pp. 161–167.

[GOG 79] J. A. Goguen and J. J. Tardo, "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specification s," *Proc. Specification of Reliable Software*, Institute of Electrical and Electronics Engineers, April 1979, pp. 170–189.

[GOG 80] J. A. Goguen, "How to Prove Algebraic Inductive Hypotheses Without Induction, With the Applications to the Correctness of Data type Implementation,"

Lecture Notes in Computer Science, 87: Proc. 5th Conf. on Automated Deduction, Les Arcs, France, Springer-Verlag, New York, July 1980, pp. 356–373.

[GUT 78a] J. V. Guttag and J. J. Horning, "The Algebraic Specification of Abstract Data Types," *Acta Informatica* **10**, pp. 27–52, 1978.

[GUT 78b] J. V. Guttag, E. Horowitz, and D. R. Musser, "Abstract Data types and Software Validation," *Communications of the ACM* **21**(12):1048–1064, Dec. 1978.

[GUT 83] J. V. Guttag, D. Kapur, and D. R. Musser, "On Proving Uniform Termination and Restricted Termination of Rewriting Systems," *SIAM Journal of Computing* **12**(1):189–214, February 1983.

[HSI 83] J. Hsiang, "Topics in Automated Theorem Proving and Program Generation," Ph.D. Thesis, Univ. of Illinois, Urbana-Champaign, Nov. 1982.

[HSI 82] J. Hsiang and N. Dershowitz, "Rewrite Methods for Clausal and Non-Clausal Theorem Proving," *Lecture Notes in Computer Science, 154: Proc. 10th EATCS intl. Collq. on Automata, Languages, and Programming*, Barcelona, Spain, Springer-Verlag, New York, July 1983, pp. 331–346.

[HUE 78] G. Huet and D. S. Lankford, "On the Uniform Halting Problem for Term Rewriting Systems," Laboratory Report 283, INRIA, Le Chesnay, France, March 1978.

[HUE 80a] G. Huet and D. C. Oppen, "Equations and Rewrite Rules: A Survey," in R. Book (Ed.), *Formal Language Theory: Perspectives and Open Problems*, Academic Press, New York, 1980, pp. 349–405.

[HUE 80b] G. Huet, "Confluente Reductions: Abstract Properties and Applications to Term Rewriting Systems," *Journal of ACM* **27**(4):797–821, October 1980.

[HUE 81] G. Huet, "A Complete Proof of Correctness of the Knuth-Bendix Completion Algorithm," *Journal of Computer and System Sciences* **23**(1):11–21, August 1981.

[HUE 82] G. Huet amd J.-M. Hullot, "Proofs by Induction in Equational Theories with Constructors," *Journal of ACM* **25**, pp. 239–266, 1982.

[HUL 80] J.-M Hullot, "Canonical Forms And Unification," *Lecture Notes in Computer Science, vol. 87: Proc. 5th Conf. on Automated Deduction*, Les Arcs, France, Springer-Verlag, New York, July 1980, pp. 318–334.

[HUL 80a] J.-M. Hullot, "A Cataloque of Canonical Term Rewriting Systems," Technical Report CSL-113, SRI Intl. Computer Science Laboratory, Menlo Park, CA, April 1980.

[ISO 86] International Standard Organization, "Information Processing Systems - Open System Interconnection - the definition of the specification language LOTOS," **ISO/DP/8807** july 1986.

[JOU 82a] J.-P Jouannaud, P. Lescanne, and F. Reinig, "Recursive Decomposition Ordering," *Proc. 2nd IFIP Workshop on Formal Description of Programming Concepts*, Garmisch-Partenkirchen. W. Germany, June 1982.

[JOU 82b] J.-P. Jouannaud and P. Lescanne, "On Multiset Ordering ," *Information Processing Letters* **15**(2), 1982.

[JOU 83] J.-P. Jouannaud, "Church-Rosser Computations with Equational Term Rewriting Systems," Technical Report, Centre de Recherche en Informatique de Nancy, Vandoeuvre-les-Nancy, France, January 1983. Submitted to *Journal of ACM.*

[JOU 84] J.-P. Jouannaud and H. Kirchner, "Completion of a Set of Rules Modulo a Set of Equations," Technical Note, SRI intl. Computer Laboratory, Menlo Park, CA, April 1984.

[KAM 80] S. Kamin and J.-J. Lévy, "Attempts for Generalising The Recursive Path Orderings," Dept. of Computer Science, Univ Illinois, Urbana-Champaign, February 1980. Unpublished manuscript.

[KAP 84a] S. Kaplan, "Conditional Rewrite Rules," in *Theoretical Computer Science*, **16**, North-Holland, 1984, pp.

[KAP 84b] S. Kaplan, "Fair Conditional Rewriting System: Unification, Termination and Confluence" Laboratory Report 410, LRI, Orsay, France, November 1984.

[KPR 83] D. Kapur and G. Sivakumar, "Experiments with the Architecture of RRL, A Rewrite Rule Laboratory," *Proc. of an NSF Workshop on the Rewrite Rule Laboratory Sept. 6-9, 1983.*

[KNU 70] D. E. Knuth and P. B. Bendix, "Simple Word Problems in Universal Algebra," in J. Leech(Ed.), *Computational Problems in Abstract Algebra*, Pergamon, Oxford, 1970, pp. 263–297.

[KOW 74] R. A. Kowalski, "Predicate Logic as a Programming Language ," *Proc. IFIP-74 congress*, North-holland, 1974, pp. 569–574.

[LAN 75a] D. S. Lankford, "Canonical Algebraic Simplification in Computational Logic," Technical Report ATP-25, Automatic Theorem Proving Project, Univ. of Texas, Austin, May 1975.

[LAN 75b] D. S. Lankford, "Canonical Inference," Technical Report ATP-32, Mathematics Dept. Univ. of Texas, Austin, December 1975.

[LAN 81] D. S. Lankford, "A Simple Explanation of Inductionless Induction," Technical Report MTP-14, Mathematics Dept., Louisiana Tech. Univ., May 1981.

[LES 83a] P. Lescanne "Computer Experiments with the REVE Term Rewriting System Generator," *Proc. 10th ACM Symp. on Principles of Programming Languages*, Austin, TX, January 1983, pp. 99–108.

[LES 83b] P. Lescanne, "Some Properties of Decomposition Ordering. A Simplification Ordering to Prove Termination of Rewriting Systems," *RAIRO Informatique Theorique/Theoretical Informatics* **16**, pp. 331–347, 1983.

[LES 84] P. Lescanne, "Uniform Termination of Term Rewriting Systems: Recursive Decomposition Ordering with Status," *proc. of 6th Colloq. on Trees in Algebra and Programming*, Bordeaux, France, Cambridge Univ. Press, March 1984.

[LIP 77] R. J. Lipton and L. Synder, "On the Halting of Tree Replacement Systems," *Proc. Conf. on Theoretical Computer Science* , Univ. of Waterloo, Waterloo, Ontario, August 1977, pp. 43–46.

[MAN 70] Z. Manna and S. Ness, "On the Termination of the Markov Algorithms," *Proc.*

3rd Hawaii Intl. Conf. on System Sciences, Honolulu, HI, January 1970, pp. 789–792.

[MAR 82] A. Martelli and U. Montanari, "An Efficient Unification Algorithm," *ACM Trans. on Programming Languages and Systems* 4(2):258–282, April 1982.

[MIL 80] R. Milner, "A Calculus of Communicating Systems," *Lecture Notes in Computer Science*, Springer-Verlag, Berlin 1980.

[MUS 80a] D. R. Musser, "Abstract Data Type Specification in the Affirm System," *IEEE Transaction on Software Engineering* 6(1):2 4–32, January 1980.

[MUS 80b] D. R. Musser, "On Proving Inductive Properties of Abstract Data Types," *Proc. 7th ACM Symp. on Principles of Programming Languages*, Las Vegas, NV, January 1980, pp. 154–162.

[NEW 42] M. H. A. Newman, "On Theories with Combinatorial Definition of 'Equivalence'," *Annals of Mathematics* 43(2):223– 243, 1942.

[PAT 78] M. S. Paterson and M. N. Wegman, "Linear Unification," *Journal of Computer and System Sciences* 16, pp. 158–167, 1978.

[PAT 81] G. E. Peterson and M. E. Stickel, "Complete Sets of Reductions for Some Equational Theories," *Journal of the ACM* 28(2):233–264, april 1981.

[PLA 78a] D. A. Plaisted, "Well-Founded Orderings for Proving Termination of Systems of Rewtite Rules," Report R-78-932, Dept. of Computer Science, Univ. of Illinois, Urbana-Champaign, July 1978.

[PLA 78b] D. A. Plaisted, "A Recursively Defined Ordering for Proving Termination of Term Rewriting Systems," Report R-78-943, Dept. of Computer Science, Univ. of Urbana-Champaign, September 1978.

[ROB 65] J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of the ACM* 12(1):23–41, January 1965.

[ROB 71] J. A. Robinson, "Computational Logic: The Unification Computation," in B. Meltzer and D. Michie (Eds.), *Machine Intelligence*, 6, Edinburgh Univ. Press, Edinburgh, Scotland, 1971, pp. 63–72.

[SCO 85] G. Scollo, G. Pappalardo, L. Logrippo, and E. Brinksma, "The OSI Transport Service and its Formal Description in LOTOS," in L. Csaba, K. Tarnay, and T. Szentivanyi, *Computer Network Usage: Recent Experiences*, North-Holland, 1986, pp 465-488

APPENDIX A

Examples of SVELDA Usage

```
/*=========================================================*/
/* Example A.1 This example demonstrates the use of the */
/* Conditional Knuth_bendix procedure to compile the    */
/* groups set of equation of Figure 3.1.                */
/* It also demonstrates the use of the validator to     */
/* prove equational theorems.                           */
/*=========================================================*/

/*=========================================================*/
/* The initial set is:                                  */
/*   []=-=>p(a,e) =-= a.                                */
/*   []=-=>p(a,i(a)) =-= e.                             */
/*   []=-=>p(p(a,b),c) =-= p(a,p(b,c)).                 */
/* where : p is the plus operator,                      */
/*         e is the identity operator                   */
/*         i is the negation operator                   */
/*=========================================================*/

/*=========================================================*/
/* While we are in the unix environmrent we type script */
/* and after starting-up svelda the execution begins    */
/*=========================================================*/

Script started on Mon Mar 16 03:57:19 1987
[1] prolog svelda_startup
C-Prolog version 1.4a
[ Restoring file svelda_startup ]

yes
| ?- svelda.

Welcome to SVELDA

**** Main Menu ****

 1.) Help
 2.) To start the ADT interpreter
 3.) To start the validator
 4.) Quit SVELDA

 Choose one of the above commands ==> 3.

**** You are within the validator, Welcome! ****

Enter a command (or helpv, or quit) ==> trace_on.

Enter a command (or helpv, or quit) ==> readf(alg).

Enter a command (or helpv, or quit) ==> ckb.

GIVEN  R1:[]=-=>p(a,e)==>a
GIVEN  R2:[]=-=>p(a,i(a))==>e
**** This formulas is not orderable under rpo ****
GIVEN  R?:[]=-=>p(p(a,b),c)==>p(a,p(b,c))
----------------------------------------------------------
Please take one of the following actions
1. Postpone the formula for the time being
2. Accept the formula as a rewrite rule in the direction shown
```

```
3. Accept the formula as a rewrite rule in the reverse direction
4. Interrupt the conditional Knuth_Bendix
-------------------------------------------------------------
|: 2.
       R3:[]=-=>p(p(a,b),c)==>p(a,p(b,c))
FROM R3,R1
R4:[]=-=>p(a,p(e,c))==>p(a,c)
FROM R3,R2
R5:[]=-=>p(a,p(b,i(p(a,b))))==>e
FROM R3,R2
R6:[]=-=>p(a,p(i(a),c))==>p(e,c)
FROM R4,R2
R7:[]=-=>p(a,i(e))==>a
FROM R6,R6
R8:[]=-=>p(e,p(i(i(a)),c))==>p(a,c)
FROM R6,R5
R9:[]=-=>p(e,p(b,i(p(i(a),b))))==>a
FROM R6,R2
R10:[]=-=>p(e,i(i(a)))==>a
REWRITE RULE : R6 FOR LEFT SIDE
FROM R9,R1
R11:[]=-=>p(e,a)==>a
FROM R9,R2
R12:[]=-=>i(i(a))==>a
R10 DISTRUCTED
R4 DISTRUCTED
R8 DISTRUCTED
FROM R9
R13:[]=-=>p(b,i(p(i(a),b)))==>a
FROM R11,R7
R14:[]=-=>i(e)==>e
R7 DISTRUCTED
FROM R6,R12
R15:[]=-=>p(i(a),p(a,c))==>c
FROM R2,R12
R16:[]=-=>p(i(a),a)==>e
FROM R15,R13
R17:[]=-=>i(p(i(a),b))==>p(i(b),a)
FROM R15,R5
R18:[]=-=>p(b,i(p(a,b)))==>i(a)
R5 DISTRUCTED
R13 DISTRUCTED
FROM R17,R15
R19:[]=-=>p(i(p(a,c)),a)==>i(c)
FROM R17,R12
R20:[]=-=>i(p(a,b))==>p(i(b),i(a))
R19 DISTRUCTED
R17 DISTRUCTED
R18 DISTRUCTED
 Stop with Success.
 Here the Complete and Consistent System.

14 : []=-=>i(e) =-= e
16 : []=-=>p(i(A),A) =-= e
15 : []=-=>p(i(A),p(A,C)) =-= C
12 : []=-=>i(i(A)) =-= A
11 : []=-=>p(e,A) =-= A
6 : []=-=>p(A,p(i(A),C)) =-= C
```

```
1 : []=-=>p(A,e) =-= A
2 : []=-=>p(A,i(A)) =-= e
3 : []=-=>p(p(A,B),C) =-= p(A,p(B,C))
20 : []=-=>i(p(A,B)) =-= p(i(B),i(A))

Enter a command (or helpv, or quit) ==> prove(i(p(i(x),i(y)))=-=p(y,i(p(i(x),

Equation being proved is:
i(p(i(x),i(y)))=-=p(y,i(p(i(x),e)))
left side being reduced:
i(p(i(x),i(y)))
--R20--> p(i(i(y)),i(i(x)))
--R12--> p(y,i(i(x)))
--R12--> p(y,x).
right side being reduced:
p(y,i(p(i(x),e)))
--R20--> p(y,p(i(e),i(i(x))))
--R14--> p(y,p(e,i(i(x))))
--R1--> p(y,i(i(x)))
--R12--> p(y,x).

Finally the equation is true in the theory

Enter a command (or helpv, or quit) ==> quit.

Back to Main Menu

**** Main Menu ****

 1.) Help
 2.) To start the ADT interpreter
 3.) To start the validator
 4.) Quit SVELDA

 Choose one of the above commands ==> 4.

 Bye Bye !!

yes
| ?- halt.

[ Prolog execution halted ]
[3] ^D
script done on Mon Mar 16 03:59:24 1987
```

```
     /*================================================*/
     /* Example A.2 : This example demonstretes the use  */
     /* of the conditional Knuth_Bendix procedure to     */
     /* compile the set of equations for lits.           */
     /*================================================*/

     /*================================================*/
     /* The initial input to the procedure is as follows */
     /*                                                  */
     /* [] ==> app(null,a) == a.                         */
     /* [] ==> app(cons(a,b),c) == cons(a,app(a,b)).     */
     /* [] ==> rev(null) == null.                        */
     /* [] ==> rev(cons(a,b)) == app(rev(b),cons(a,null)).
     /* [] ==> rev(rev(b)) == b.                         */
     /*================================================*/

     /*================================================*/
     /* While in unix environment we type script then    */
     /* we start_up svelda and begin the completion      */
     /* process.                                         */
     /*================================================*/

Script started on Mon Mar 16 04:15:50 1987
[1] prolog svelda_startup
C-Prolog version 1.4a
[ Restoring file svelda_startup ]

yes
| ?- svelda.

Welcome to SVELDA

**** Main Menu ****

 1.) Help
 2.) To start the ADT interpreter
 3.) To start the validator
 4.) Quit SVELDA

 Choose one of the above commands ==> 3.

**** You are within the validator, Welcome! ****

Enter a command (or helpv, or quit) ==> trace_on.

Enter a command (or helpv, or quit) ==> readf(liste).

Enter a command (or helpv, or quit) ==> ckb.

GIVEN  R1:[]==>app(null,a)==>a
GIVEN  R2:[]==>app(cons(a,b),c)==>cons(a,app(b,c))
GIVEN  R3:[]==>rev(null)==>null
GIVEN  R4:[]==>rev(cons(a,b))==>app(rev(b),cons(a,null))
GIVEN  R5:[]==>rev(rev(b))==>b
FROM R5,R4
R6:[]==>rev(app(rev(b),cons(a,null)))==>cons(a,b)
FROM R6,R5
R7:[]==>rev(app(b,cons(a,null)))==>cons(a,rev(b))
```

```
R6 DISTRUCTED
 Stop with Success.
 Here the Complete and Consistent System.

4 : []=-=>rev(cons(A,B)) =-= app(rev(B),cons(A,null))
2 : []=-=>app(cons(A,B),C) =-= cons(A,app(B,C))
1 : []=-=>app(null,A) =-= A
3 : []=-=>rev(null) =-= null
5 : []=-=>rev(rev(B)) =-= B
7 : []=-=>rev(app(B,cons(A,null))) =-= cons(A,rev(B))

Enter a command (or helpv, or quit) ==> quit.

Back to Main Menu

**** Main Menu ****

 1.) Help
 2.) To start the ADT interpreter
 3.) To start the validator
 4.) Quit SVELDA

 Choose one of the above commands ==> 4.

 Bye Bye !!

yes
| ?- halt.

[ Prolog execution halted ]
[3] ^D
script done on Mon Mar 16 04:16:24 1987
```

```
    /*=============================================*/
    /* Example A.3 : This example demonstrate the  */
    /* use of the conditional Knuth-Bendix proc.   */
    /* in the conditional case.                    */
    /* It also demonstrate the rejection of non-   */
    /* Feasible formulas.                          */
    /*=============================================*/

Script started on Mon Mar 16 05:23:00 1987
[1] prolog svelda_startup
C-Prolog version 1.4a
[ Restoring file svelda_startup ]

yes
| ?- svelda.

Welcome to SVELDA

**** Main Menu ****

 1.) Help
 2.) To start the ADT interpreter
 3.) To start the validator
 4.) Quit SVELDA

 Choose one of the above commands ==> 3.

**** You are within the validator, Welcome! ****

Enter a command (or helpv, or quit) ==> trace_on.

Enter a command (or helpv, or quit) ==> readf(evod).

Enter a command (or helpv, or quit) ==> ckb.

GIVEN  R1:[]=-=>even(zero)==>true
GIVEN  R2:[]=-=>even(succ(zero))=-=>false
GIVEN  R3:[]=-=>even(succ(succ(x)))=-=>even(x)
GIVEN  R4:[even(x)-=-true]=-=>odd(x)=-=>false
GIVEN  R5:[even(x)-=-false]=-=>odd(x)=-=>true
**** The following formula is discarded as being non-feasible ****
FROM R4,R5 [even(x)-=-true,even(x)-=-false]=-=>false=-=true
 Stop with Success.
 Here the Complete and Consistent System.

2 : []=-=>even(succ(zero)) =-= false
1 : []=-=>even(zero) =-= true
3 : []=-=>even(succ(succ(X))) =-= even(X)
4 : [even(X)-=-true]=-=>odd(X) =-= false
5 : [even(X)-=-false]=-=>odd(X) =-= true

Enter a command (or helpv, or quit) ==> quit.

Back to Main Menu

**** Main Menu ****

 1.) Help
```

```
 2.) To start the ADT interpreter
 3.) To start the validator
 4.) Quit SVELDA

 Choose one of the above commands ==> 4.

 Bye Bye !!

yes
| ?- halt.

[ Prolog execution halted ]
[3] ^D
script done on Mon Mar 16 05:28:14 1987
```

```
     /*=================================================*/
     /* Example A.4: This example gives another case of    */
     /* using the conditional Knuth-Bendix procedure on a  */
     /* a set of conditional axioms.                       */
     /* This set of axioms is the one of Figure 3.3        */
     /*=================================================*/

Script started on Mon Mar 16 06:00:17 1987
[1] prolog svelda_startup
C-Prolog version 1.4a
[ Restoring file svelda_startup ]

yes
| ?- svelda.

Welcome to SVELDA

**** Main Menu ****

 1.) Help
 2.) To start the ADT interpreter
 3.) To start the validator
 4.) Quit SVELDA

 Choose one of the above commands ==> 3.

**** You are within the validator, Welcome! ****

Enter a command (or helpv, or quit) ==> readf(int).

Enter a command (or helpv, or quit) ==> sort_on.

Enter a command (or helpv, or quit) ==> ckb.

GIVEN  R1:[]=->succ(pred(x))==>x
GIVEN  R2:[]=->pred(succ(x))==>x
GIVEN  R3:[]=->leq(zero,zero)==>true
GIVEN  R4:[]=->leq(zero,pred(zero))==>false
GIVEN  R5:[leq(zero,x)-=-true]=->leq(zero,succ(x))==>true
GIVEN  R6:[leq(zero,x)-=-false]=->leq(zero,pred(x))==>false
**** This formulas is not orderable under rpo ****
GIVEN  R?:[]=->leq(succ(x),y)==>leq(x,pred(y))
-------------------------------------------------------
Please take one of the following actions
1. Postpone the formula for the time being
2. Accept the formula as a rewrite rule in the direction shown
3. Accept the formula as a rewrite rule in the reverse direction
4. Interrupt the conditional Knuth_Bendix
-------------------------------------------------------
|: 2.
       R7:[]=->leq(succ(x),y)==>leq(x,pred(y))
FROM R7,R1
R8:[]=->leq(pred(x),pred(y))==>leq(x,y)
**** This formulas is not orderable under rpo ****
GIVEN  R?:[]=->leq(pred(x),y)==>leq(x,succ(y))
-------------------------------------------------------
Please take one of the following actions
1. Postpone the formula for the time being
```

```
2. Accept the formula as a rewrite rule in the direction shown
3. Accept the formula as a rewrite rule in the reverse direction
4. Interrupt the conditional Knuth_Bendix
------------------------------------------------------------
|: 2.
       R9:[]-=>leq(pred(x),y)==>leq(x,succ(y))
R8 DISTRUCTED
 Stop with Success.
 Here the Complete and Consistent System.

5 : [leq(zero,X)-=-true]-=>leq(zero,succ(X)) =-= true
6 : [leq(zero,X)-=-false]-=>leq(zero,pred(X)) =-= false
3 : []-=>leq(zero,zero) =-= true
1 : []-=>succ(pred(X)) =-= X
2 : []-=>pred(succ(X)) =-= X
4 : []-=>leq(zero,pred(zero)) =-= false
7 : []-=>leq(succ(X),Y) =-= leq(X,pred(Y))
9 : []-=>leq(pred(X),Y) =-= leq(X,succ(Y))

Enter a command (or helpv, or quit) ==> quit.

Back to Main Menu

**** Main Menu ****

 1.) Help
 2.) To start the ADT interpreter
 3.) To start the validator
 4.) Quit SVELDA

 Choose one of the above commands ==> 4.

 Bye Bye !!

yes
| ?- halt.

[ Prolog execution halted ]
[3] ^D
script done on Mon Mar 16 06:05:36 1987
```

```
/*================================================*/
/* Example A.5: This example demonstrates the use */
/* of the ADT interpreter in case of overloaded   */
/* operation. The specification used here is      */
/* a definition of the queue of integer with the  */
/* operation if_then_else being overloaded.       */
/*                                                */
/* This is the specification of the queue of int- */
/* ger in internal form. We can see that the      */
/* operation if_then_else is overloaded this exa- */
/* show the capability of the sort checker.       */
/*                                                */
/* sort int => if_then_else(bool,int,int).        */
/* sort int => zero.                              */
/* sort queue => if_then_else(bool,queue,queue).  */
/* sort queue => new.                             */
/* sort queue => rem(queue).                      */
/* sort queue => add(int,queue).                  */
/* sort int   => first(queue).                    */
/* sort bool  => empty(queue).                    */
/* sort bool  => true.                            */
/* sort bool  => false.                           */
/*                                                */
/* [] =-=> if_then_else(true,X,Y) ==> X.          */
/* [] =-=> if_then_else(false,X,Y) ==> Y.         */
/* [] =-=> rem(new)      ==> new.                 */
/* [] =-=> rem(add(M,Q)) ==>                      */
/*       if_then_else(empty(Q),new,add(M,rem(Q))). */
/* [] =-=> first(new)       ==> zero.             */
/* [] =-=> first(add(M,Q)) ==>                    */
/*       if_then_else(empty(Q),M,first(Q)).       */
/* [] =-=> empty(new)       ==> true.             */
/* [] =-=> empty(add(M,Q)) ==> false.             */
/*================================================*/

Script started on Tue Mar 17 04:53:29 1987
[1] prolog svelda_startup
C-Prolog version 1.4a
[ Restoring file svelda_startup ]

yes
| ?- svelda.

Welcome to SVELDA

**** Main Menu ****

 1.) Help
 2.) To start the ADT interpreter
 3.) To start the validator
 4.) Quit SVELDA

 Choose one of the above commands ==> 2.

**** You are within the ADT interpreter, Welcome! ****

Enter a command (or helpi or quit) ==> helpi.
```

```
**** ADT interpreter commands Menu ****
readf(fn) : read the specification in the file fn
permut_on : set the rewriting engine in the permutative mode
permut_off : reset the rewriting engine in normal mode (default)
eval(expr) : evaluate the expression expr
sort_of(expr) : calculate the sort of expr
sort_on : evaluation using sort checking
sort_off : evaluation without sort checking
trace_on : set the tracer on
trace_off : reset the tracer off (default)

Enter a command (or helpi or quit) ==> read(queue).

Enter a command (or helpi or quit) ==> sort_on.

Enter a command (or helpi or quit) ==> eval(rem(add(zero,new))).

Expression being evaluated is:
rem(add(zero,new))
After sort checking and evaluation we get:
sort: queue
term: new

Enter a command (or helpi or quit) ==> eval(first(add(zero,new))).

Expression being evaluated is:
first(add(zero,new))
After sort checking and evaluation we get:
sort: int
term: zero

Enter a command (or helpi or quit) ==> quit.

Back to Main Menu

**** Main Menu ****

 1.) Help
 2.) To start the ADT interpreter
 3.) To start the validator
 4.) Quit SVELDA

 Choose one of the above commands ==> 4.

 Bye Bye !!

yes
| ?- halt.

[ Prolog execution halted ]
[3] ^D
script done on Tue Mar 17 04:55:36 1987
```

```
/*================================================*/
/* Example A.6: This example demonstrates the     */
/* use of the ADT interpreter in the permutative  */
/* mode. Here in constract with example one we    */
/* choose to leave the rule that expresses asso-  */
/* ciativity as an unoriented rule so it is used  */
/* by the rewriting engine as a permutative rule  */
/* . The example use the 4 rules used in sedction*/
/* 4.5.2.                                         */
/*                                                */
/* 1:[] =-=> plus(X,zero) ==> X.                  */
/* 2:[] =-=> plus(X,i(X)) ==> zero.               */
/* 3:[] =-=> plus(X,Y) ==> plus(Y,Z).             */
/* 4:[] =-=>                                      */
/*      plus(plus(X,Y),Z) =-= plus(X,plus(Y,Z))   */
/*                                                */
/* Note that the integer label associated with    */
/* each rule is added atomaticaly by ADT          */
/* interpreter.                                   */
/*================================================*/

Script started on Tue Mar 17 05:21:42 1987
[1] prolog svelda_startup
C-Prolog version 1.4a
[ Restoring file svelda_startup ]

yes
| ?- svelda.

Welcome to SVELDA

**** Main Menu ****

 1.) Help
 2.) To start the ADT interpreter
 3.) To start the validator
 4.) Quit SVELDA

 Choose one of the above commands ==> 2.

**** You are within the ADT interpreter, Welcome! ****

Enter a command (or helpi or quit) ==> readf(pint).

Enter a command (or helpi or quit) ==> trace_on.

Enter a command (or helpi or quit) ==> permut_on.

Enter a command (or helpi or quit) ==> eval(plus(plus(x,y),i(x))).

Expression being evaluated is:
plus(plus(x,y),i(x))
--R3--> plus(plus(y,x),i(x))
--R4--> plus(y,plus(x,i(x)))
--R2--> plus(y,zero)
--R1--> y.

Enter a command (or helpi or quit) ==> eval(plus(i(x),plus(y,x))).
```

```
Expression being avaluated is:
plus(i(x),plus(y,x))
--R3--> plus(i(x),plus(x,y)).
--R4--> plus(plus(i(x),x),y)
--R3--> plus(plus(x,i(x)),y)
--R2--> plus(zero,y)
--R3--> plus(y,zero)
--R1--> y.

Enter a command (or helpi or quit) ==> quit.

Back to Main Menu

**** Main Menu ****

 1.) Help
 2.) To start the ADT interpreter
 3.) To start the validator
 4.) Quit SVELDA

 Choose one of the above commands ==> 4.

 Bye Bye !!

yes
| ?- halt.

[ Prolog execution halted ]
[3] ^D
script done on Tue Mar 17 05:24:46 1987
```

APPENDIX B

Program Listing

```
            /*=============================================*/
            /* Implementation of the help System and comands */
            /*=============================================*/
svelda :- nl,write('Welcome to SVELDA'),nl,nl,
          start_up.

start_up :- repeat,
            menu1(CH),
            (CH == 1 -> main_help;
             CH == 2 -> start_interpreter;
             CH == 3 -> start_validator;
             CH == 4 -> nl,nl, write(' Bye Bye !!'),nl,nl;
             otherwise -> nl,nl,write('**** Bad choice!!, Enter 1., 2., 3., o
             CH == 4,!.

menu1(CH) :- nl,nl,write('**** Main Menu ****'),
             nl,nl,
             write(' 1.) Help'),nl,
             write(' 2.) To start the ADT interpreter'),nl,
             write(' 3.) To start the validator'),nl,
             write(' 4.) Quit SVELDA'),nl,nl,
             write(' Choose one of the above commands ==> '),
             see(user),read(CH),seen,!.

start_interpreter :- nl,write('**** You are within the ADT interpreter, Welco
                     nl,start_int.

start_int :- repeat,
                     nl,nl,write('Enter a command (or helpi or quit) ==> '),
                     see(user),read(ANS),seen,execute(ANS),ANS == quit,!.

execute(helpi) :- help_int,!.
execute(helpv) :- help_val,!.
execute(quit) :- nl,nl,write('Back to Main Menu'),nl,!.
execute(ANS) :- call(ANS),!.
execute(_)  :- nl,nl,write('**** Wrong Command ****'),!.

help_int :- nl,nl,write('**** ADT interpreter commands Menu ****'),nl,
            write('readf(fn) : read the specification in the file fn'),nl,
            write('permut_on : set the rewriting engine in the permutative mo
            write('permut_off : reset the rewriting engine in normal mode (de
            write('eval(expr) : evaluate the expression expr'),nl,
            write('sort_of(expr) : calculate the sort of expr'),nl,
            write('sort_on : evaluation using sort checking'),nl,
            write('sort_off : evaluation without sort checking'),nl,
            write('trace_on : set the tracer on'),nl,
            write('trace_off : reset the tracer off'),nl,!.

start_validator :- nl,write('**** You are within the validator, Welcome! ****
                   start_v.

start_v :- repeat,
           nl,write('Enter a command (or helpv, or quit) ==> '),
           see(user),
           read(ANS),seen,
           execute(ANS),ANS == quit,!.

help_val :- nl,write('**** Validator commands Menu ****'),nl,
```

```
            write('readf(fn). : read the specification in the file fn'),
          nl,write('ckb. : run the conditional Knuth_bendix'),nl,
            write('prove(Eq). : prove the equation in the theory of the ex
            write('termination. : Prove the termination of the existing TR
            write('trace_on. : set the tracer on (defualt)'),nl,
            write('tracer_off : reset the tracer off'),nl,!.

main_help :- nl,write('**** Svelda Commands Menu ****'),nl,nl,
             write('help_int. : Display the ADT interpreter command menu'),nl
             write('help_val. : Display the validator commands menu'),nl,
             write('help_oth. : Display the remaining commands'),nl,!.

help_oth :- nl,write('**** More about commands ****'),nl,
            nl,write('freeze(fn). : save the specification in the file fn'),n
            write('thaw(fn). restore the specification in the file fn'),nl,!.

permut_on :- remove(permut(X)),
             assert(permut(on)),!.
permut_off :- remove(permut(X)),
              assert(permut(off)),!.

sort_on :- remove(sor(X)),
           assert(sor(on)),!.
sort_off :- remove(sor(X)),
           assert(sor(off)),!.

trace_on :- remove(trac(X)),
            assert(trac(on)),!.
trace_off :- remove(trac(X)),
            assert(trac(off)),!.

freeze(Fn) :- save(Fn),!.
thaw(Fn) :- reconsult(Fn),!.
```

```
?- op(125,fx,'sort').
?- op(126,xfy,'=>').
?- [rwc,'unify_type'].
            /*=========================================*/
            /* The evaluation function prolog implem.  */
            /*=========================================*/


eval(V) :- sor(on),check_expr_type(V,VT),!,rwc(V,VN),
           write('Expression being evaluated is:'),nl,
           write(V),nl,
           write('After evaluation we get:'),nl,
           get_type(VN,VNT),
           write('sort:'),write(VNT),nl,
           write('term:'),write(VN),nl,!.
eval(V) :- sor(off),rwc(V,VN),
              write('Expression being evaluated is:'),nl,
              write(V),nl,
              !.

            /*=========================================*/
            /* The type checker prolog implemetation   */
            /*=========================================*/


check_expr_type(P,PT) :- P =.. [Fn|Args],
                         check_args_type(Args,T_args),
                         PT =.. [Fn|T_args],
                         functor(PT,Fn,N),
                         functor(Xp,Fn,N),
                         sort T => Xp,
                         unify1(PT,Xp).

check_expr_type(P,PT) :- write('**** Erreur de Type dans ****'),nl,
                            write(P),nl,
                            !,fail.

check_args_type([],[]) :- !.
check_args_type([X|Y],[Xt|Yt]) :- get_type(X,Xt),
                                  check_args_type(Y,Yt).

get_type(X,Xt) :- atomic(X),
                  sort Xt => X.
get_type(X,Xt) :- var(X).
get_type(X,Xt) :- X =.. [Fn|Args],
                  check_args_type(Args,T_args),
                  Xf =.. [Fn|T_args],
                  functor(Xf,Fn,N),
                  functor(Xp,Fn,N),
                  sort Xt => Xp,
                  unify1(Xf,Xp).
```

```
?- op(245,xfy,':').
?- op(256,xfy,'=-=').
?- op(256,xfy,'==>').
?- op(256,xfy,'-=-').
?- op(255,xfy,'=-=>').


?- [rwengine,normalize,utility,rpo].

   /*=========================================*/
   /*           Conditional Knuth-Bendix      */
   /*=========================================*/
ckb :-
                        knuth_bendix(Ind,Eq,X,Y),fail.
ckb :-
                      write(' Stop with Success. '),nl,
                      write(' Here the Complete and Consistent System.'),nl,
                      nl,print_regle(I,Eq,X,Y),fail.
ckb  :- !.

knuth_bendix(Ind,Eq,X,Y)   :-
                        traite(E,P,Q,En,PN,QN),fail.

knuth_bendix(Ind,Eq,X,Y) :- consider_postponed(Ind,Eq,X,Y),
                           knuth_Bendix(I,Eq1,X1,Y1).

knuth_bendix(Ind,Eq,X,Y) :-
                        ==>(Ind:Eq=-=>X,Y), /*calcul des paires critique pou
                        not(visited(Ind)),
                        assert(visited(Ind)),
                        pair(Ind,Eq,X,Y),
                        knuth_bendix(I,Eq1,X1,Y1).


knuth_bendix(Ind,Eq,X,Y) :- consider_large(Ind,Eq,X,Y),
                           knuth_bendix(I,Eq1,X1,Y1).



traite(Eq,P,Q,Eqn,Pn,Qn) :-
                        pc(Eq,Ind,P,Ix,Q),remove(pc(Eq,Ind,P,Ix,Q)),
                        rw(P,PN,0),
                        rw(Q,QN,0),
                        change(Eq,Eqn,PN,Pn,QN,Qn),
                        orient(1,Eqn,Ind,Pn,Ix,Qn).

remove(X) :- retract(X),!.

pair(Ind,Eq,X,Y) :-
                ==>(Ix:Eq2=-=>Z,T),
                Ix =< Ind,
                superpose(Ind,X,Y,Ix,Z,T,I),
                ( I = 1,cal_cpair(Ind,Eq,X,Y,Ix,Eq2,Z,T),fail
                  ;
                  I = 2,cal_cpair(Ix,Eq2,Z,T,Ind,Eq,X,Y),fail).
pair(_,_,_,_) .
```

```
superpose(Ind,X,Y,Ix,Z,T,I) :-
                Z =.. [Fz|Zargs],
                occur_in(Fz,X),I is 1,!.

superpose(Ind,X,Y,Ix,Z,T,I) :-
                X =.. [Fx,Xargs],
                occur_in(Fx,Z),I is 2,!.

occur_in(F,Term) :-
                retractall(pred(_)),
                cal_pred(Term),
                pred(F),!.

cal_pred(Term) :-
                not(atom(Term)),
                Term =.. [Fn|Args],
                assertz(pred(Fn)),
                cal_pred_list(Args).
cal_pred(_) :- !.

cal_pred_list([]) :- !.
cal_pred_list([H|T]) :-
                cal_pred(H),
                cal_pred_list(T).

cal_cpair(Ind,Eq1,X,Y,Ix,Eq2,Z,T) :-
                str(X,_),
                sout(ST),
                remove(sout(ST)),
                retractall(s(_,_)),
                ST \== Z,
                manip(Ind,Ix,Eq2,Z,T,Eq2n,Zn,Tn),
                unify(ST,Zn),
                substitute(Tn,ST,X,Xnew),
                check_map(Xnew,Xs),
                check_map(Y,Ys),
                append(Eq1,Eq2n,Eq12),
                check_map_list(Eq12,Eq12s),
                retractall(s(_,_)),
                add_or_discard(Ind,Ix,Eq12s,Xs,Ys).

consider_postponed(Ind,Eq,X,Y) :- post(Eq,Ind,X,I,Y),
                                  remove(post(Eq,Ind,X,I,Y)),
                                  rw(X,Xn,0),
                                  rw(Y,Yn,0),
                                  orient(2,Eq,Ind,Xn,I,Yn),!.

consider_large(Ind,Eq,X,Y) :- large(Eq,Ind,X,I,Y),
                              remove(large(Eq,Ind,X,I,Y)),
                              test(Eq,Ind,X,Y),!.
```

```
        /*********************************************/
        /*          The Rewriting Engine             */
        /*********************************************/

rw(V,VN,R)     :- nonvar(V),not(atomic(V)),
                  !,
                  V =.. [Fn|Args],
                  rwl(Args,NewArgs,R),
                  Vl =.. [Fn|NewArgs],
                  rw2(Ind,Vl,VN,R).
rw(V,VN,R)     :- rw2(Ind,V,VN,R),!.

rwl([],[],R)  :- !.
rwl([H|T],[B|Rs],R) :- rw(H,B,R),rwl(T,Rs,R).

rw2(I,X,Y,0)     :- nonvar(X),rr0(I:Eq=-=>X,Z),
                    rwp(Eq,R),
                    !,
                    rw(Z,Y,R).
rw2(I,X,Y,2)     :- nonvar(X),rr2(I:Eq=-=>X,Z),
                    rwp(Eq,R),
                    !,
                    rw(Z,Y,R).
rw2(I,X,Y,3)     :- nonvar(X),rr3(I:Eq=-=>X,Z),
                    rwp(Eq,R),
                    !,
                    rw(Z,Y,R).
rw2(_,X,X,R)     :- !.

rwp([],R)          :- !.
rwp([U=-=V|Rs],R) :- rw(U,Un,R),
                     rw(V,Vn,R),
                     Un == Vn,
                     rwp(Rs,R).
```

```
            /*=========================================*/
            /* This is the prolog implementation of the */
            /* recursive path ordering.                 */
            /*=========================================*/
rpo(X,Y):-
    qrpo(X,Y),not(qrpo(Y,X)),!.

qrpo(X,Y) :-
          atomic(X),atomic(Y),X==Y,!.
qrpo(X,Y) :-
not(atomic(X)),atomic(Y),not(varb(Y)),!.
qrpo(X,Y) :-
      not(atomic(X)),varb(Y),!.
qrpo(X,Y) :-
             not(atomic(X)),not(atomic(Y)),
             X=..[Fx|Argx],Y=..[Fy|Argy],
             gt(Fx,Fy),
             setrpo(X,Argy),!.
qrpo(X,Y) :-
             not(atomic(X)),not(atomic(Y)),
             X=..[Fx|Argx],Y=..[Fy|Argy],
             Fx==Fy,
             multisetrpo(Argx,Argy),!.
qrpo(X,Y) :-
             not(atomic(X)),not(atomic(Y)),
             X=..[Fx|Argx],
             setqrpo(Argx,Y),!.

setrpo(X,[]) :- !.
setrpo(X,[Hy|Ty]) :-
                     rpo(X,Hy),
                     setrpo(X,Ty).

setqrpo([],Y) :- fail,!.
setqrpo([Hx|Tx],Y) :-
                      (qrpo(Hx,Y);setqrpo(Tx,Y)),!.

multisetrpo(Argx,Argy) :-
                        residue(Argx,Argy,Rest),
                        mrpo(Argx,Rest),!.

mrpo(_,[]) :- !.
mrpo(Argx,[Hr|Tr]) :- listmrpo(Argx,Hr),
                      mrpo(Argx,Tr),!.

listmrpo([],Y) :- fail,!.
listmrpo([Hx|Tx],Y) :- (rpo(Hx,Y);listmrpo(Tx,Y)),!.

residue(_,[],[]).
residue(Argx,[Hy|Ty],Rest) :- member(Hy,Argx),
                              residue(Argx,Ty,Rest),!.
residue(Argx,[Hy|Ty],[Hy|Rest]) :- residue(Argx,Ty,Rest),!.

member(X,[X|_]) :- !.
member(X,[_|T]) :- member(X,T).

subterm(X,Y) :- X=..[Fx|Argx],
                occur_in(Argx,Y),!.
```

```
occur_in(Argx,Y) :- member(Y,Argx),!.
occur_in([Hx|Tx],Y) :- not(atomic(Hx)),
                       Hx=..[H|T],
                       (occur_in(T,Y);occur_in(Tx,Y)),!.
```

```
        /*****************************************/
        /*   Normalizing The Rewriting System    */
        /*****************************************/


orient(Flag,Eq,Ind,Pn,Ix,Qn) :- Pn == Qn,print_dpair(Ind,_,Ix,_),!.
orient(Flag,Eq,Ind,Pn,Ix,Qn) :- rpo(Pn,Qn),analyse(2,Eq,Pn,Qn,Ind,Ix),!.
orient(Flag,Eq,Ind,Pn,Ix,Qn) :- rpo(Qn,Pn),analyse(3,Eq,Pn,Qn,Ind,Ix),!.
orient(1,Eq,Ind,Pn,Ix,Qn) :- assertz(post(Eq,Ind,Pn,Ix,Qn)),!.
orient(2,Eq,Ind,Pn,Ix,Qn) :-  nl,
                  write('**** This formulas is not orderable under rpo ****'),
                  nl,nl,
                  print_choice(Eq,Pn,Qn,Ind,Ix),
                  see(user),
                  read(C),
                  seen,
                  analyse(C,Eq,Pn,Qn,Ind,Ix)
                 ,!.

analyse(1,Eq,Pn,Qn,Ind,Ix) :- assertz(post(Eq,Ind,Pn,Ix,Qn)),!.
analyse(2,Eq,Pn,Qn,Ind,Ix) :- trans(Eq,Pn,Qn),
                      m_a_j(Id,Eq,K,L),
                      nb_regle(I),assertz((==>(I:Eq=-=>Pn,Qn))),
                      print_cpair(I,Eq,Ind,Pn,Ix,Qn),
                      !.
analyse(3,Eq,Pn,Qn,Ind,Ix) :- trans(Eq,Qn,Pn),
                      m_a_j(Idx,Eq,Kx,Lx),
                      nb_regle(I),assertz((==>(I:Eq=-=>Qn,Pn))),
                      print_cpair(I,Eq,Ind,Qn,Ix,Pn),
                      !.
analyse(4,_,_,_,_,_)    :- write('Stop with failure. '),nl,call(abort).

trans(Eq,Pl,Ql)       :-
                      transfer1(_,_,_,_),
                      transfer2(_,_,_,_),
                      cal(In),
                      chan(In,Eq,Eql,Pl,P,Ql,Q,2),
                      readh(h),
                      retractall(rr0(_:_=-=>_,_)),
                      retractall(rr3(_:_=-=>_,_)),
                      chan(In,Eq,Eq2,Pl,P2,Ql,Q2,3),
                      readh(h),
                      chan(In,Eq,Eq3,Pl,P3,Ql,Q3,0),
                      readh(h),
                      nl.

m_a_j(Id,Eq,K,L) :-
                      rrl(Id:Eq=-=>K,L),
                      rw(K,Kn,3),
                      test2(Id,Eq,K,L,Kn),fail.
m_a_j(_,_,_,_) :- !.

test2(Id,Eq,K,L,Kn) :- K == Kn,
                      rw(L,Ln,2),
                      rewrite_left(Id,K,L,Ln),
                      remove(==>(Id:Eq=-=>K,L)),
                      addto(Id,Eq,K,Ln),
                      chan(Id,Eq,Eql,K,Kl,Ln,Lnl,0),
```

```
                    readh(h),!.
test2(Id,Eq,K,L,Kn) :- remove(==>(Id:Eq=-=>K,L)),
                    asserta(pc(Eq,Id,Kn,0,L)),
                    !.

rewrite_left(Id,K,L,Ln) :- L == Ln,!.
rewrite_left(Id,K,L,Ln) :- write('REWRITE RULE : R'),write(Id),
                           write('FOR LEFT SIDE'),
                           nl,!.

addto(I,Eq,K,Ln) :-     visited(I),assertz(==>(I:Eq=-=>K,Ln)),!.
addto(I,Eq,K,Ln) :-     asserta(==>(I:Eq=-=>K,Ln)),!.

transfer1(Id,Eq,X,Y) :- retractall(rr1(_:_=-=>_,_)),
                    ==>(Id:Eq=-=>X,Y),
                    putin(rr1(Id:Eq=-=>X,Y)),fail.
transfer1(_,_,_,_) :- !.

transfer2(Id,Eq,X,Y) :- retractall(rr2(_:_=-=>_,_)),
                    rr0(Id:Eq=-=>X,Y),
                    putin(rr2(Id:Eq=-=>X,Y)),fail.
transfer2(_,_,_,_) :- !.

retractall(X) :- retract(X),fail.
retractall(_) :- !.

cal(In) :- nb_regle(I),retractall(nb_regle(I)),In is I + 1,
           assert(nb_regle(In)), !.

putin(X) :- assertz(X),!.

print_cpair(Ind,Eq,0,Pn,0,Qn) :- write('GIVEN  '),write('R'),write(Ind),
                           write(':'),
                           write(Eq),write('=-=>'),
                           write(Pn),write('==>'),write(Qn),
                           nl,!.
print_cpair(Ind,Eq,I,Pn,0,Qn) :- write('FROM '),write('R'),write(I),nl,
                           write('R'),write(Ind),write(':'),
                           write(Eq),write('=-=>'),
                           write(Pn),write('==>'),write(Qn),
                           nl,!.
print_cpair(Ind,Eq,I,Pn,J,Qn) :- write('FROM '),write('R'),write(I),
                           write(','),write('R'),write(J),nl,
                           write('R'),write(Ind),write(':'),
                           write(Eq),write('=-=>'),
                           write(Pn),write('==>'),write(Qn),
                           nl,!.

print_dpair(Ind,_,0,_) :- write('R'),write(Ind),
                           write(' DISTRUCTED '),
                           nl,!.
print_dpair(_,_,_,_)   :- !.
```

```
   /*=========================================*/
   /* This clause is used to printout the     */
   /* the rewrite rules.                      */
   /*=========================================*/

print_regle(Ind,Eq,X,Y) :-
                    ==>(Ind:Eq=-=>X,Y),
                    chan(Ind,Eq,Eqn,X,Xn,Y,Yn,0),
                    write(Ind),write(' : '),write(Eqn),
                    write('=-=>'),write(Xn),write(' =-= '),write(Yn),
                    nl.
print_regle(_,_,_,_) :- !.

   /*=========================================*/
   /* substitute(New,Old,Val,NewVal) :        */
   /* substitute Old by New in Val giving     */
   /* NewVal.                                 */
   /*=========================================*/

substitute(New,Old,Old,New) :- !.
substitute(New,Old,Val,Val) :-
                        atomic(Val),!.
substitute(New,Old,Val,NewVal) :-
                        Val =.. [Fn|Args],
                        subst_args(New,Old,Args,NewArgs),
                        NewVal =.. [Fn|NewArgs],!.

   /*=========================================*/
   /* subst_args(New,Old,L1,L2) : substitute  */
   /* all occurrences of Old by New in the    */
   /* list L1 giving the liste L.             */
   /*=========================================*/

subst_args(_,_,[],[]) :-  !.
subst_args(New,Old,[Old|Args],[New|NewArgs]) :-
                        subst_args(New,Old,Args,NewArgs).
subst_args(New,Old,[Arg|Args],[NewArg|NewArgs]) :-
                        substitute(New,Old,Arg,NewArg),
                        subst_args(New,Old,Args,NewArgs).

   /*=========================================*/
   /* readf(F): read the file F.              */
   /*=========================================*/

readf(F) :-
                        see(F),repeat,read(T),pro(T),seen,!.

   /*=========================================*/
   /* pro(T): return if end of file or assert */
   /* T as declaration fact or as a feasible  */
   /* critical pair depending on whether T is */
   /* a signature or a formula.               */
   /*=========================================*/

pro(T) :-
                        end_of_file_mark(T),!.
pro(Clause) :- Clause = gt(X,Y),assert(gt(X,Y)),fail.
pro(Clause) :- Clause = varb(X),assert(varb(X)),fail.
```

```
pro(Clause) :-
                        Clause = =-=(Eq=-=>X,Y),assertz(pc(Eq,0,X,0,Y)),fai

end_of_file_mark(end_of_file).

   /*=======================================*/
   /* retractall(X): retract all the rules  */
   /* with the predicate name X from the    */
   /* knowledge base.                       */
   /*=======================================*/

retractall(X) :-
                        retract(X),fail.
retractall(_) :- !.

   /*=======================================*/
   /* str(E,_),sit(L1,L2): look for all the */
   /* the subterms of the term E which are  */
   /* strictly contained in E.              */
   /*=======================================*/

str(E,E) :-             atomic(E),!.
str(E,_) :-
                        E =.. [Op|R],
                        add_if_not_in(E),
                        pph(R),
                        sit(R,La).

sit([],[]).
sit([X1|L1],X1) :-
                        str(X1,X),
                        sit(L1,Y).

add_if_not_in(E) :- sout(E),!.
add_if_not_in(E) :- assertz(sout(E)),!.

   /*=======================================*/
   /* pph(L) : assert the memeber of the list */
   /* L in knowledge base.                  */
   /*=======================================*/

pph([]) .
pph([H|T]) :-
                        atomic(H),pph(T),!.
pph([H|T]) :-
                        add_if_not_in(H),pph(T).

   /*=======================================*/
   /* The following procedure are performs some */
   /* operations on lits.                   */
   /*=======================================*/

check_map_list([],[]) :- !.
check_map_list([X-=-Y|T],[Xm-=-Ym|Tc]) :-
                        check_map(X,Xm),
                        check_map(Y,Ym),
                        check_map_list(T,Tc),!.
```

```
map_list([],[]):-          !.
map_list([X|L],[Y|M]) :-
                           check_map(X,Y),
                           map_list(L,M),!.

check_map(Y,Ys) :-
                           not(varb(Y)),atomic(Y), Ys = Y, !.
check_map(Y,Ys) :-
                           varb(Y), sub(Y,Ys),!.
check_map(Y,Ys) :-
                           Y =.. [Fy|YArgs],
                           map_list(YArgs,YsArgs),
                           Ys =.. [Fy|YsArgs],
                           !.

sub(Y,Ys) :- s(Y,Z),
             check_map(Z,Ys).

sub(Y,Y) :- !.
   /*=========================================*/
   /* gensym(Prefix,Var) : genrate new symbol  */
   /* composed of Prefix and an integer giving */
   /* Var.                                     */
   /*=========================================*/

gensym(Prefix,Var) :-
                        var(Var),atomic(Prefix),
                        gett(Prefix,N),
                        N1 is N+1,
                        concat(Prefix,N1,Var).

gett(Prefix,0).

   /*=========================================*/
   /* concat(N1,N2,N) : concat N1 and N2      */
   /* given the result in asccii mode in N    */
   /*=========================================*/

concat(N1,N2,N) :-
                        name(N1,Ls1),
                        name(N2,Ls2),
                        append(Ls1,Ls2,Ls),
                        name(N,Ls).


append([],L,L) :- !.
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

   /*=========================================*/
   /* change(Eq,Eqc,X,Y,Z,T) : renames varia-*/
   /* bles in Eq,X,Y and in Eqc,Z,T to elimi-*/
   /* nate common varible in the formula      */
   /* composed by Eq,X,Y and the formula      */
   /* composed of Eqc,Z, and T.               */
   /* =========================================*/

change(Eq,Eqc,X,Y,Z,T) :-
                        X =.. [Fn|Args],
```

```
                        process(Fn,Args,F,NArgs),
                        Y =.. [F|NArgs],
                        Z =.. [Fn1|Args1],
                        process(Fn1,Args1,F1,NArgs1),
                        T =.. [F1|NArgs1],
                        process_list(Eq,Eqc),
                        gett(Prefix,Pr),remove(gett(Prefix,Pr)),
                        Nxt is Pr + 1,
                        assertz(gett(Prefix,Nxt)),!.

process_list([],[]) :- !.
process_list([X-=-Y|T],[Xc-=-Yc|Tc]) :-
                        X =.. [Fn|Args],
                        process(Fn,Args,F,NArgs),
                        Xc =.. [F|NArgs],
                        Y =.. [Fn1|Args1],
                        process(Fn1,Args1,F1,NArgs1),
                        Yc =.. [F1|NArgs1],
                        process_list(T,Tc),!.

process(F,[],T,X) :-   check1(F,T), X = [],   !.
process(F,[X|R],T,[Xn|Rn]) :-
                        check1(X,Xn),
                        process(F,R,T,Rn).

check1(Q,S) :-
                        varb(Q),gensym(Q,S),assert_varb_ifnotin(S),!.
check1(Q,S) :-
                        not(varb(Q)),atomic(Q),Q = S,!.
check1(Q,S) :-
                        Q =.. [F|Ar],
                        process(F,Ar,Fx,NAr),
                        S =.. [Fx|NAr].

assert_varb_ifnotin(S) :- varb(S),!.
assert_varb_ifnotin(S) :- assert(varb(S)),!.


manip(Ind,Ix,Eq,STU,Z,Eqn,STn,Zn) :-
                          Ind == Ix,
                          change(Eq,Eqn,STU,STn,Z,Zn),!.
manip(Ind,Ix,Eq,STU,Z,Eqn,STn,Zn) :-
                          Eqn = Eq,
                          STn = STU,
                          Zn = Z,!.


remove(X) :- retract(X),!.

genvar(Prefix,Var) :-
                    var(Var),atomic(Prefix),
                    getvar(Prefix,Var),!.


getvar(Nl,N) :-
                  name(Nl,Lsl),
                  first_of(Lsl,X),
                  Y is X - 32,
```

```
                name(N,[Y]),!.


first_of([X|R],X) :- !.

      /*=========================================*/
      /* These are some utility function used by  */
      /* the Conditional Knuth_Bendix procedure   */
      /*=========================================*/

chan(I,E,En,X,Y,Z,T,K) :-
            X =.. [Fn|Args],
            pcess(Fn,Args,F,NArgs),
            Y =.. [F|NArgs],
            Z =.. [F1|Args1],
            pcess(F1,Args1,F2,NArgs1),
            T =.. [F2|NArgs1],
            pcess_list(E,En),
            tell(h), write('rr'),write(K),write('('),write(I),write(':'),
            write(En),write('=-=>'),
            write(Y),write(','),write(T),write('.'),told,!.

pcess_list([],[]) :- !.
pcess_list([X-=-Y|T],[Xc-=-Yc|Tc]) :-
                     X =.. [Fn|Args],
                     pcess(Fn,Args,F,NArgs),
                     Xc =.. [F|NArgs],
                     Y =.. [Fn1|Args1],
                     pcess(Fn1,Args1,F1,NArgs1),
                     Yc =.. [F1|NArgs1],
                     pcess_list(T,Tc),!.

pcess(F,[],T,X)  :- chck1(F,T), X = [],!.
pcess(F,[X|R],T,[Xn|Rn]) :- chck1(X,Xn),
                            pcess(F,R,T,Rn),!.

chck1(Q,S) :-  varb(Q), genvar(Q,S),!.
chck1(Q,S) :- not(varb(Q)),atomic(Q),Q = S,!.
chck1(Q,S) :- Q =.. [F|Ar],
              pcess(F,Ar,Fx,NAr),
              S =.. [Fx|NAr],!.
readh(H) :-   see(H),read(T),assertz(T),seen,!.

        /***********************************************/
        /* This procedure is used to calculate the   */
        /* number of variable in a term X.           */
        /***********************************************/



calnbvar(X) :-
            retractall(nbvarb(_)),assertz(nbvarb(0)),
            X =.. [Fn|Args],
            process1(Args).

calnbvar(_,_) :- !.

process1([]) :- !.
```

```
process1([X|R]) :- check2(X),
                   process1(R).

check2(Q) :-  varb(Q),cal(Q,In),!.
check2(Q) :- not(varb(Q)),atomic(Q),!.
check2(Q) :- Q =.. [F|Ar],
             process1(Ar).

cal(Q,In) :- nbvarb(I),retractall(nbvarb(I)), In is I + 1,
          assert(nbvarb(In)),!.

       /*=========================================*/
       /* This procedure decides if the formula   */
       /* should be added to the feasible critical */
       /* pairs set or to discarded or to added as */
       /* a to the knowledge base as a large      */
       /* formula.                                */
       /*=========================================*/

add_or_discard(Ind,Ix,Eq,Xs,Ys) :- calnbvar(Xs),
                        feasible(Eq),
                        decision(Ind,Ix,Eq,Xs,Ys),!.

decision(Ind,Ix,Eq,Xs,Ys) :- nbvarb(I),maxcomp(Il),
                  ( I < Il; Ind < Ix) ,
                   assertz(pc(Eq,Ind,Xs,Ix,Ys)),!.
decision(Ind,Ix,Eq,Xs,Ys) :- nbvarb(I),maxcomp(Il),
                   I >= Il
                   ,assertz(pcdisc(Eq,Ind,Xs,Ix,Ys)),!.

      /*=========================================*/
      /* This procedure is used to check for the */
      /* feasibility of the premisses.           */
      /*=========================================*/

feasible([]) :- !.
feasible([U-=-V|T]) :-
                free_var(U,V,Eq),
                rwp([Eq],0),
                feasible(T).

free_var(U,V,Eq) :- U =.. [Fn|Args],
                    pcess(Fn,Args,F,NArgs),
                    Uf =.. [F|NArgs],
                    V =.. [Fn1|Argsl],
                    pcess(Fn1,Argsl,Fl,NArgsl),
                    Vf =.. [Fl|NArgsl],
                    tell(e),write(Uf),write('-=-'),write(Vf),write('.'),told,
                    see(e),read(Eq),seen,!.


   /*=========================================*/
   /* This the unification procedure that     */
   /* incorporate the occur_check test        */
   /*=========================================*/

occur_check(X,Y) :-
```

```
                    get(Y,Yu),
                    not(varb(Yu)),not(atomic(Yu)),
                    Yu =.. [Fn|Args],
                    member1(X,Args),
                    !.

occur_check(X,Y) :- get(X,Xu),
                    not(varb(Y)),not(atomic(Y)),
                    varb(Xu),
                    Y =.. [Fn|Args],
                    member1(Xu,Args),!.


occur_check(X,Y) :-
                    not(varb(Y)),
                    get(X,Xu),
                    not(varb(Xu)),atomic(Xu),
                    Xu \== Y,!.
        /*====================================*/
member1(X,[X|Y]) :- !.
member1(X,[V|Y]) :- not(atomic(V)),V =.. [Fn|Args],member1(X,Args).
member1(X,[Z|Y]) :- member1(X,Y).
        /*====================================*/

get(Y,Yu) :- not(varb(Y)),not(atomic(Y)),
             !,
             Y =.. [Fn|Args],
             get1(Args,Argsu),
             Y1 =.. [Fn|Argsu],
             get2(Y1,Yu).

get(Y,Yu) :- get2(Y,Yu),!.
get1([],[]) :- !.
get1([H|T],[B|R]) :- get(H,B),get1(T,R).

get2(X,Y) :- varb(X),
             s(X,Z),!,
             get(Z,Y).
get2(X,X) :- !.

        /*====================================*/
unify(X,Y) :- varb(X),
              !,
              not(occur_check(X,Y)),
              assert(s(X,Y)),!.
unify(X,Y) :- varb(Y),
              !,
              not(occur_check(Y,X)),
              assert(s(Y,X)),!.
unify(X,X) :- atomic(X),!.
unify(X,Y) :- X =.. [Fn|XArgs],
              Y =.. [Fn|YArgs],
              unify_list(XArgs,YArgs).
unify_list([],[]).
unify_list([X|Y],[X1|Y1]) :- unify(X,X1),
                             unify_list(Y,Y1).
```

```
        /*=======================================*/
        /* This procedure is used in case of     */
        /* an unorderable formula to give the user */
        /* the actions that he can select.       */
        /*=======================================*/
print_choice(Eq,Pn,Qn,Ind,Ix) :- nl,
                        print_cpair('?',Eq,Ind,Pn,Ix,Qn),
                        write('-----------------------------------------
                        write('Please take one of the following actions'),n
                        write('1. Postpone the formula for the time being')
                        nl,
                        write('2. Accept the formula as a rewrite rule in t
                        write('3. Accept the formula as a rewrite rule in t
                        write('4. Interrupt the conditional Knuth_Bendix'),
                        write('-----------------------------------------
                        !.
```

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA